



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

AUTOMATICKÉ VYHODNOCOVÁNÍ STUDENTSKÝCH PROJEKTŮ V PYTHONU

AUTOMATIC ASSIGNMENT OF STUDENT PROJECTS IN PYTHON

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE

AUTHOR

JURAJ KYSEL'

VEDOUCÍ PRÁCE

SUPERVISOR

Doc. RNDr. PAVEL SMRŽ, Ph.D.

BRNO 2018

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačové grafiky a multimédií

Akademický rok 2017/2018

Zadání bakalářské práce

Řešitel: **Kysel' Juraj**

Obor: Informační technologie

Téma: **Automatické vyhodnocování studentských projektů v Pythonu**
Automatic Assignment of Student Projects in Python

Kategorie: Informační systémy

Pokyny:

1. Seznamte se se systémy typu Automatic Tutor se zaměřením na automatické vyhodnocování studentských projektů.
2. Získejte a zpracujte přehled nástrojů pro odhalování nedostatků v kódu dynamických jazyků a možností doporučování lepších postupů z hlediska výkonnosti a čistoty kódu.
3. Navrhněte a implementujte systém, který dokáže hodnotit studentské projekty v Pythonu a navrhopat zlepšení.
4. Vyhodnoťte vytvořený systém na sadě projektů z předmětu Skriptovací jazyky.
5. Vytvořte stručný plakát prezentující práci, její cíle a výsledky.

Literatura:

- dle dohody s vedoucím

Pro udělení zápočtu za první semestr je požadováno:

- funkční prototyp řešení

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese

<http://www.fit.vutbr.cz/info/szz/>

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Smrž Pavel, doc. RNDr., Ph.D., UPGM FIT VUT**

Datum zadání: 1. listopadu 2017

Datum odevzdání: 16. května 2018

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačové grafiky a multimédií
Ládv. ob. Brno, Božetěchova 2



doc. Dr. Ing. Jan Černocký
vedoucí ústavu

Abstrakt

Táto práca sa zameriava na automatické vyhodnocovanie študentských kódov v programovacom jazyku Python. Dôraz je predovšetkým kladený na spätnú väzbu, ktorá má študentov usmerniť v správnych praktikách písania pythonovského kódu. V teoretickej časti sa práca zaoberá existujúcimi riešeniami a taktiež celkovo skúma problematiku vyhodnocovania kódov a výuku správnych programovacích návykov. V praktickej časti sú získané informácie použité na navrhnutie systému, ktorý začínajúcich programátorov upozorní na chyby a snaží sa im vnuknúť správne konštrukcie v podobe tzv. pythonovských idiémov. Ďalej práca popisuje jednotlivé časti systému, jeho vstupy a výstupy. Ako prípadová štúdia bol systém nasadený pre podporu výuky predmetu Skriptovacie jazyky na FIT VUT v Brně v akademickom roku 2017/2018. Práca zhrňuje skúsenosti z tohoto nasadenia a hodnotí prednosti aj problémy spracovaného riešenia.

Abstract

This work focuses on automatic evaluation of student codes in Python programming language. Emphasis is placed primarily on feedback, to guide students in the correct Python code writing practices. In the practical part, the information obtained is used to design a system that alerts beginning programmers to errors and tries to teach them the right structures in the form of python idioms. Further, the thesis describes individual parts of the system, its inputs and outputs. As a case study, the system was put in place to support the teaching of the subject Scripting Languages at the FIT VUT in Brno in the academic year 2017/2018. The thesis summarizes the experience of this deployment and evaluates the advantages and problems of the processed solution.

Klíčové slová

Python, automatické vyhodnocovanie projektov, idiém, spätná väzba, webová aplikácia

Keywords

Python, automatic project evaluation, idiom, feedback, web application

Citácia

KYSEL, Juraj. *Automatické vyhodnocování studentských projektů v Pythonu*. Brno, 2018. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Doc. RNDr. Pavel Smrž, Ph.D.

Automatické vyhodnocování studentských projektů v Pythonu

Prehlásenie

Prehlasujem, že som túto bakalársku prácu vypracoval samostatne pod vedením pána Doc. RNDr. Pavla Smrža, Ph.D. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

.....

Juraj Kysel
11. mája 2018

Podakovanie

Chcel by som sa poďakovať Doc. RNDr. Pavlovi Smržovi, Ph.D. za vedenie, ochotu a poskytnutú pomoc v podobe konzultácii a informačných zdrojov. Ďalej by som rád poďakoval svojej rodine a priateľom za poskytnutú morálnu podporu.

Obsah

1	Úvod	3
2	Programátorské chyby a ich detekcia	5
2.1	Chyby nezávislé na programovacím jazyku	6
2.2	Syntaktické chyby	7
2.3	Sémantické chyby	7
2.4	Logické chyby	8
2.5	Nesprávne praktiky	8
2.5.1	Programový idióm	8
2.5.2	Chyby programového štýlu	9
2.6	Chyby v čase kompilácie	9
2.7	Chyby pri behu programu	9
3	Chyby v jazyku Python	11
3.1	Chyby syntaxu	11
3.2	Chyby sémantiky	12
3.3	Chyby indentácie	12
3.4	Nesprávne praktiky	13
3.4.1	Chyby programového štýlu	13
4	Automatizácia výuky v programovaní	16
4.1	E-learning	16
4.1.1	Code School	17
4.1.2	Codecademy	17
4.2	Analýza kódu	17
4.2.1	Python Tutor	18
4.2.2	Learn Python	18
5	Návrh	20
5.1	Vstup	20
5.2	Spracovanie zdrojového kódu a prípravenie na beh testov	21
5.3	Spustenie testov na detekciu chýb	22
5.4	Prípravenie dát so spätnou väzbou	22
5.5	Výstup	22
5.6	Webové rozhranie	22
6	Implementácia	24
6.1	Webové rozhranie	24

6.2	Vstup	25
6.3	Spracovanie zdrojového kódu	26
6.4	Testy na detekciu chýb	28
6.4.1	Testy idiómov	28
6.4.2	Testy projektov	30
6.4.3	Testy štýlu	30
6.4.4	Iné testy	31
6.5	Záznam o chybe	31
6.5.1	Logovanie	31
6.6	Zobrazenie spätnej väzby	32
6.7	Deployment	32
7	Testovanie	34
7.1	1. projekt	34
7.2	2. projekt	35
7.3	3. projekt	36
7.4	4. projekt	36
7.5	5. projekt	37
7.6	6. projekt	39
7.7	7. projekt	40
8	Závěr	42
	Literatúra	43
	Prílohy	45
A	Návrh	46
B	Grafické užívateľské rozhranie	47

Kapitola 1

Úvod

Dopyt po programátoroch pribúdajúcimi rokmi stále rastie, a tak rastú aj nároky na ich výučbu. Jazyk ako taký, sa dá najefektívnejšie naučiť jeho používaním. To platí aj pre programovacie jazyky. Dôležitým faktorom je, aby študent dostal kvalitnú spätnú väzbu, pri najlepšom okamžitú. To je pri veľkom počte študentov na jedného vyučujúceho takmer nemožné. Preto sa začali vyvíjať nástroje na automatické vyhodnocovanie študentských zdrojových kódov a prác [15]. Tieto nástroje majú suplovať prítomnosť vyučujúceho, aby sa veľký počet študentov súčasne mohol rýchlo a samostatne zdokonaľovať [5].

Mnohí študenti sa neučili Python ako prvý jazyk. Konštrukcie, ktoré boli vyhovujúce v inom jazyku, nemusia platiť aj v Pythone. Tým pádom treba študentov na tieto chyby upozorniť. Tieto konštrukcie môže byť obtiažne nájsť v obecnom kóde, nakoľko existuje veľmi veľa foriem, v akých sa môžu vyskytnúť. Pri zadaných príkladoch, ktoré majú študenti vypracovať, je to inak. U nich je známe, ako by malo optimálne riešenie vyzeráť. Takýmto testovaním je možné študenta naviesť na správnu cestu. Ďalšou pomôckou pri testovaní je použitie *assert* testov. Tie sa dajú využiť na overenie funkcionality zadaných príkladov, ale nie sú úplne efektívne pri hľadaní konkrétnych nevhodných konštrukcií. Využitie kombinácie oboch spomínaných metód môže viesť ku efektívnemu nájdeniu nesprávnych konštrukcií v zdrojovom kóde študenta.

Táto práca sa zaoberá vývojom systému pre automatické vyhodnocovanie študentských prác v jazyku Python. Aplikácia vyhodnotí študentský kód po štylistickej stránke, ako aj overí korektnosť použitých konštrukcií v rámci programovacieho jazyka Python. Výsledkom je okamžitá spätná väzba, kde sú študentovi ukázané jeho chyby a predložený príklad na zlepšenie, ako aj odkazy na literatúru, kde sa môže o danej problematike dočítať viac. Hlavný zámer systému nespočíva v hľadaní syntaktických chýb, ale o navedenie študenta na používanie správnych a efektívnych konštrukcií, ktoré jazyk Python ponúka.

Cieľom tejto práce je:

- zoznámenie čitateľa s častými chybami začínajúcich programátorov
- objasnenie možností detekcie takýchto chýb
- predstavenie výsledného systému, ktorý vznikol v rámci tejto práce
- zhrnutie výsledkov z nasadenia vytvoreného systému v predmete Skriptovacie jazyky v akademickom roku 2017/2018

Kapitola 2 popisuje vo všeobecnosti všetky chyby, ktoré sa vyskytujú pri programovaní, ako aj spôsob, ako ich riešiť a predchádzať im. Kapitola 3 už o týchto chybách hovorí

iba v rámci jazyka Python. Rovnako, ako im predíť. V kapitole 4 práca popisuje niektoré spôsoby, ktorými sa dá výuka programovacieho jazyka Python automatizovať. Sú tu predstavené reálne výukové prostriedky.

Nasledujúce kapitoly popisujú už samostnú prácu. Kapitola 5 popisuje návrh aplikácie v štádiu čisto po naštudovaní literatúry. Výsledná implementácia teda nemusí byť s návrhom totožná. Tú je možné nájsť v kapitole 6. V nej je do podrobna popísaná štruktúra výsledného systému a jeho častí. V kapitole 7 je popísaný spôsob testovania vyvinutého systému. V rámci tejto bakalárskej práce bol systém nasadený do prevádzky ako podporná výuková pomôcka predmetu *ISJ* v akademickom roku 2017/2018. Kapitola popisuje výsledky testov jednotlivých projektov.

Kapitola 2

Programátorské chyby a ich detekcia

V rámci vylepšenia výuky programovacích jazykov je vhodné sa najprv zamerať na začiatkových programátorov. Konkrétnejšie na ich chyby. Štúdiom týchto chýb sme následne schopní zamerať sa na najproblémovjšie časti daného programovacieho jazyka. To nám môže pomôcť zefektívniť a vylepšiť nasledujúcu výučbu.

Takéto dáta sa rozhodli analyzovať dvaja páni z univerzity v Kente, ktorí vo svojej práci: *37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data* [1] skategorizovali najčastejšie chyby začínajúcich programátorov. Dáta boli zozbierané na Java zdrojových kódach od 250 000 študentov z celého sveta, čo je prijateľne veľká vzorka. Z celkového počtu 37 miliónov kompilácii kódov bolo približne 47% neúspešných. Najčastejšie vyskytované chyby boli sémantické a typové. Ak by sme mali chyby zasadiť do časového rozmedzia procesu učenia, tak najprv sa najčastejšie vyskytujú syntaktické chyby. Ako sa ale študent časom ztotožňuje so syntaxom daného jazyka, tieto chyby miznú a do popredia sa dostávajú sémantické chyby. Štúdia popisuje aj konkrétne chyby jazyka Java, ich frekvenciu ako aj čas na ich odstránenie.

Programátorské chyby teda môžeme rozdeliť na:

- Chyby nezávislé na programovacom jazyku
- Chyby závislé od programovacieho jazyka - tie sa avšak dajú ešte bližšie špecifikovať. Tieto chyby môžeme ďalej deliť na:
 - Syntaktické chyby
 - Sémantické chyby
 - Logické chyby
 - Nesprávne praktiky - táto kategória nepopisuje priamo chyby, ktoré kladú prekážku vo funkčnosti programu, ale častokrát môžu byť dôvodom, prečo takéto chyby vznikajú.

Existuje však aj iné delenie, ktoré sa zaoberá tým, kedy sa chyby vyskytnú:

- Chyby v čase kompilácie
- Chyby pri behu programu

2.1 Chyby nezávislé na programovacím jazyku

Úplní začiatčníci si často komunikáciu medzi nimi a počítačom prirovnávajú ku komunikácii v klasickej konverzácii. Tak to ale v programovaní nefunguje. O tomto hovorí článok: *Language-Independent Conceptual 'Bugs' in Novice Programming* [10]. Táto chybná úvaha sa dá ale ďalej rozdeliť na tri triedy:

- Chyba paralelizmu - jeho podstata spočíva v tom, že študent trpí domnienkou, že rôzne riadky programu sú aktívne v rovnakom čase. Resp. počítač ich vykonáva v rovnakom čase. Ako príklad je uvedená podmienka `if`, ktorá sa podľa študenta vyhodnotí, nech je program v akomkoľvek štádiu. To znamená, že ak na začiatku programu kontrolujeme premennú oproti konkrétnej hodnote a niekde ďalej v kóde premenná túto hodnotu dosiahne, podmienka sa vyhodnotí kladne. Túto odpoveď zvolilo osem z pätnástich opýtaných študentov. Ich zmýšľanie by sa dalo popísať reálnym príkladom. 'Ak chceš ísť do obchodu, zaveziem ťa'. Ak by sa študent rozhodol o hodinu ísť do obchodu, je dosť možné, že by som ho stále zaviezol. Presne tu nastáva rozdiel v komunikácii medzi ľuďmi a počítačom.
- Chyba intencionality¹ - jedná sa o chybu z dôvodu, že študent vytvára program za nejakým jednoznačným a priamočiarym účelom. To znamená, že nedáva pozor na ostatné okolnosti, ktoré sa môžu vyskytnúť. Keď sa nejaká podmienka vyhodnotí záporne ale on očakáva kladný výsledok, tak to je pre neho odklonenie od kurzu a touto variantou sa nezaobera. Článok popisuje príklad, kde dali študentom nakresliť na papier obdĺžnik a nahlas majú popisovať, čo robia. Častokrát to skončilo nepresným popisom ako 'Nakreslím štyri čiary aby uzavreli priestor medzi sebou'. Takémuto popisu ale chýba napríklad dĺžka čiar, o koľko stupňov sa treba otočiť a podobne.
- Chyba egocentrizmu - tieto chyby sú opakom intencionálnych. Študent podvedome očakáva od počítača podobnú reakciu ako od človeka. To znamená, že pri vynechaní čo i len najmenšieho ale dôležitého detailu čaká, že si počítač túto skutočnosť domyslí. Pri chybe si častokrát študent povie 'ja som to tak nemyslel'. Preto je veľmi dôležité počítaču prese špecifikovať, čo od neho chceme.

Tu nastáva rozpor medzi tým, v čo študenti veria a čo robia. Študenti popierajú, že by mal počítač nejakú umelú inteligenciu, ktorá si dokáže domýšľať ich zámer pri programovaní. Už od prvých hodín je im vštepovaná myšlienka, že počítače sú hlúpe a urobia iba to, čo im ľudia povedia. No častokrát konajú opačne. Článok tento jav pripisuje tomu, že študent si vie komunikáciu medzi programátorom a počítačom pripísať ku komunikácii medzi dvojmi ľuďmi. Preto sa častokrát pri vyskytnutom probléme inštinktívne vráti ku zapoužívaným praktikám z tejto komunikácie.

Ďalším problémom, ktorý stojí za zmienku, je kolidovanie pomenovaní. Kľúčové slová² boli v programovacích jazykoch zámerne pomenované tak, aby reprezentovali akcie, ktoré si môžeme priradiť k tým v reálnom svete. Pomenovanie vlastných premenných tak, aby reprezentovali objekty s ktorými sa pracuje je dobrý štýl zapisovania. Zbehlý programátor vie, ktorým pomenovaniam sa vyhnúť, aby nekolidovali s kľúčovými slovami. Problém nastane medzi začiatčníkmi, ktorí tieto medze ešte nepoznajú.

¹Intencionalita - zameranie ľudského snaženia na určitý cieľ.

²Slová vyčlenené v syntaxi programovacieho jazyka, ktoré majú špeciálny význam. Tieto slová sa nesmú použiť ako identifikátor (pomenovanie premennej).

2.2 Syntaktické chyby

Tieto chyby sa vyskytujú v každom jazyku, no v každom jazyku vznikajú za iných podmienok. To je z dôvodu, že syntax³ je jedinečný pre každý programovací jazyk.

Čo je to vlastne syntaktická chyba? Je to chyba v zdrojovom kóde programu. Programátor napísal niečo, čo počítač nenašiel ako prijateľnú syntaktickú konštrukciu v danom programovacom jazyku. Tu je možné vidieť rozdiel medzi komunikáciou medzi počítačom a človek oproti komunikácii dvoch ľudí. V klasickej konverzácii si človek dokáže domyslieť isté súvislosti, aj keď rozprávajúci urobil vo výklade nejakú malú chybu. Počítače ale nedokážu 'prečítať' správu, ktorá nie je perfektne napísaná. Zdrojový kód, ktorý obsahuje syntaktickú chybu nie je možné preložiť, tým pádom ani spustiť.

Nájdenie týchto chýb je z pohľadu toho, kto píše kód, jednoduché. Postará sa o to kompilátor⁴. Keď kompilátor nevie preložiť všetky syntaktické konštrukcie, vyhodí chybu a zastaví kompiláciu. Kompilátor úspešne preloží kód, až keď dokáže rozpoznať všetky syntaktické konštrukcie. Keďže sa o nájdenie chyby postará kompilátor, na programátorovi ostáva už iba chybu odstrániť. Začínajúcim programátorom sa odporúča používať syntaktické príručky, ak je potreba.

Jednoduchším spôsobom, ako sa vyhnúť alebo až predísť syntaktickým chybám je používaním softwarového IDE⁵. IDE zvyčajne obsahujú editor zdrojového kódu, prostriedky na kompiláciu a spustenie programu a debugger⁶. Novšie obsahujú inteligentné dopĺňanie kódu na základe programovacieho jazyka a taktiež automatickú kontrolu syntaxa jazyka v reálnom čase. Táto funkcia je veľmi nápomocná pre začiatočníkov ale samotné IDE môže byť niekedy pre nich príliš komplikované a mäťúce.

2.3 Sémantické chyby

Tieto chyby plynú zo zlého použitia výrazov, typu premenných, operácii, použitia operácií v zlom poradí, neexistujúcich premenných a podobne. To znamená, že takýto program je po syntaktickej stránke správny. Tým pádom ich kompilátor neoznačí ako chybné. Problém nastáva až pri behu programu. Tu sa spomínané chyby prejavia. Ako príklad by som uviedol jednoduché delenie 5/0. Po syntaktickej stránke je tento výraz správny, nakoľko delím číslo iným číslom. Kompilátor označí výraz ako správny. No pri spustení programu sa objaví chyba delenia nulou.

Oprava týchto chýb nie je taká priamočiara, ako pri chybe syntaxu. Chyba syntaxu je chyba gramatiky programovacieho jazyka. Sémantická chyba avšak súvisí s tým, čo programátor myslel svojim zápisom. Čo ale napísal, nemalo žiadny zmysel. Častokrát majú začínajúci programátori problém tieto chyby odhaliť, pretože sú na problém zameraní z jedného uhlu pohľadu. Dobrou pomôckou môže byť krokovanie programu, ktoré umožní odhaliť chybu. Sémantické chyby majú väčšinou svoj typ. Teda programátor vie, akej chyby sa do-

³Súbor pravidiel definujúcich kombinácie postupnosti znakov, ktoré sú následne považované za správne v danom programovacom jazyku.

⁴Kompilátor alebo kompilujúci program alebo prekladač je program, ktorý dokáže preložiť zdrojový kód napísaný v niektorom programovacom jazyku do iného programovacieho jazyka, najčastejšie do strojového kódu.

⁵Softwarová aplikácia obsahujúca nástroje na efektívne vyvíjanie počítačových programov alebo aplikácií.

⁶Počítačový program, ktorý slúži na vyhľadávanie chýb v zdrojových kódoch.

pustil. Pri delení 5/0 by sa objavila chyba *Division by zero* a programátor vie, kam by mal svoju pozornosť pri oprave zdrojového kódu upriamiť.

2.4 Logické chyby

Tieto chyby sa vyskytnú už pri samotnom zadávaní úlohy alebo jej pochopení. Prejavia sa avšak až na úplnom konci vývoja programu. Výsledný kód je bezchybný a robí presne to, čo programátor chcel, ale jeho interpretácia problému je nesprávna. V jednoduchosti sa teda jedná o zlé pochopenia problému. Logická chyba môže nastať zlým úsudkom, použitím zlého modelu alebo nepochopením zákazníka. Rovnako ako pri sémantických chybách, kompilátor takúto chybu nedetekuje. Programátor si pri najlepšom chybu všimne pri overovaní správnosti výsledkov. Môže to avšak zájsť aj do takého extrému, že sa problém prejaví napríklad až pri predávaní produktu zákazníkovi.

Čím skôr je túto chybu možné detekovať, tým lepšie. Kľúčom je korektne pochopiť daný problém. Ak programátor nemá úplne jasno v probléme, výskyt takýchto chýb sa môže zvýšiť. Ak má začínajúci programátor pochybnosti o správnosti pochopenia problému, je veľmi vhodné poradiť sa s niekým. Bolo by optimálne, keby to bol skúsenejší programátor, no už len diskutovanie o probléme s ďalším začínajúcim programátorom môže zvýšiť šance o správne pochopenie problému.

2.5 Nesprávne praktiky

Nejedná sa v pravom slova zmysle o chyby. V programovaní ide v prvom rade o rýchlosť a efektivitu. Čo i len sekunda navyše je v programovaní veľmi dlhý čas. Takže ak sa dá beh programu urýchliť už iba tým, akým štýlom program napíšeme, možno stojí za uvažovanie sa nad touto témou zamyslieť. Na nesprávne praktiky nám nepoukáže kompilátor ani žiadna chybová správa. Prejavia sa až pri behu programu. Najčastejšou známkou je dlhý beh programu alebo pamäťová náročnosť. Pri jednoduchých programoch je to pominuteľné a začínajúci programátori, ktorí vytvárajú krátke programy, si to ani nevšimnú. Ale ak sa zlým praktikám naučia už pri učení programovacieho jazyka, tieto chyby sa mnohokrát znásobia pri vytváraní väčších projektov, ktorým sa budú v budúcnosti venovať.

Nesprávnymi praktikami sa teda rozumie nevyužitie plného potenciálu, ktorý daný programovací jazyk ponúka. Jedná sa o nesprávny alebo neefektívny zápis riešenia nejakého problému. Výsledný program funguje ako má. Bezchybný je aj po logickej stránke. Do tejto kategórie patrí aj štylistická stránka zdrojového kódu. Čiže zápis, formátovanie a čitateľnosť.

2.5.1 Programový idióm

Rovnako ako ľudský jazyk, programovacie jazyky majú idiómy, ktoré sa dajú označiť za obvyklý spôsob, akým je možné vyjadriť nejakú myšlienku. Uviedol by som príklad, kde sa fráza '*Som hladný*' v angličtine používa ako '*I am hungry*'. Vo francúzskom jazyku môžeme povedať doslova to isté, ako '*Je suis affamé*' ale ľudia tak vôbec nehovoria. Gramaticky je to správne, ale nie je to idióm. Namiesto toho ľudia hovoria '*J ai faim*', čo v doslovnom preklade znamená '*Mám hlad*'. Do angličtiny sa to preloží teda ako '*I have hunger*', čo zasa v angličtine nikto nepoužíva - nie je to idióm.

Podobný prístup sa dá aplikovať aj pri programovaní. V rámci programovania je idióm možné definovať ako zaužívanú konštrukciu, akou možno naprogramovať určitú úlohu v da-

nom programovacím jazyku. Hlavnou časťou procesu učenia nového programovacieho jazyka by malo byť stotožnenie sa s jeho idiómami.

O použití idiómov pre konzistenciu zdrojového kódu pojednáva kniha: *The Practice of Programming* [7, p. 10]. V nej sa hovorí aj o tom, ako je používanie idiómov rozšírené a tým pádom je kód lepšie čitateľný a pochopiteľnejší pre iného programátora.

Idiomy však netreba brať ako zákony, ktoré sa nesmú porušiť. Je ich potreba brať s rezervou a používať s uvážením vzhľadom na konkrétnu situáciu.

Programovať idiomatically teda znamená využívať konštrukcie, ktoré sú jedinečné pre daný programovací jazyk. Niektoré jazyky sú na idiomy bohaté. Napríklad jazyk Perl ich obsahuje množstvo, kde na druhej strane jazyk Java ich veľkým množstvom neprekypuje.

2.5.2 Chyby programové štýlu

Hovoríme o takzvaných kozmetických chybách. Programový štýl je súbor pravidiel, ktoré sa používajú pri písaní zdrojového kódu. Tieto pravidlá zjednocujú písanie kódu medzi programátormi, čiže sa rôznym programátorom jednoduchšie číta kód po niekom inom. Kód má jednotný štýl, a tým sa zlepšuje prehľadnosť a znižuje prípadná chybovosť. Medzi prvky štýlu patrí napríklad odriadkovanie, vynechávanie prázdnych riadkov na konkrétnych miestach za účelom prehľadnosti. Ďalej sa sem zaraďuje dĺžka riadkov, formátovanie textu, používanie komentárov a ich formát, štýl pomenovania objektov. Dokumentácia kódu je taktiež veľmi dôležitá, nakoľko umožňuje takpovediac náhľad do mysle programátora, ktorý kód vyvíjal a tým zlepšuje čitateľnosť a orientáciu v kóde.

Tu sa taktiež nejedná o chyby detekovateľné počítačom. Ich používanie ale zlepšuje opätovné používanie zdrojového kódu.

2.6 Chyby v čase kompilácie

Používanejší termín v programovaní je *Compilation error*. Jedná sa o chyby, ktoré sa vyskytnú pri preklade zdrojového kódu napísaného programátorom do kódu strojového. Vyskytnú sa teda vtedy, ak počítač nevie 'prečítať', čo mu programátor napísal. Patria medzi ne syntaktické chyby alebo statické sémantické chyby.

V závislosti na programovacom jazyku sa sem môže zaradiť aj takzvaný *Linker error*. Ten sa vyskytne pri spájaní všetkých potrebných súborov, funkcií alebo knižníc, ktoré sú potrebné na beh spúšťaného programu. Chceme napríklad použiť nejakú špeciálnu funkciu, ktorá sa nachádza v externej knižnici. Pokým túto knižnicu nenaimportujeme, budeme dostávať chybu od linkera⁷.

2.7 Chyby pri behu programu

Používanejší termín v programovaní je *Runtime error*. Preklad zdrojového kódu na strojový prebehne bez chyby. Pri spustení programu však nastane chyba. Tá sa nemusí vyskytnúť hneď, ale kludne až po pár sekundách behu programu. To môže byť z dôvodu nedostatku pamäte. Dôvodom nedostatku pamäte môže byť fakt, že program vyžaduje kontinuálne viac a viac pamäte RAM na svoj beh až do bodu, kedy ďalšie pridelenie operačný systém nedovolí z bezpečnostných dôvodov. Nekonečné zacyklenie, alokovanie pamäte alebo nedealokovanie

⁷Program, ktorý vezme jeden alebo viac objektových súborov a spojí ich do jedného spustiteľného

sú príčinami týchto chýb. Ďalším príkladom môže byť prístup ku prvku mimo generovaného pola. Takýmto chybám hovoríme aj dynamické sémantické chyby.

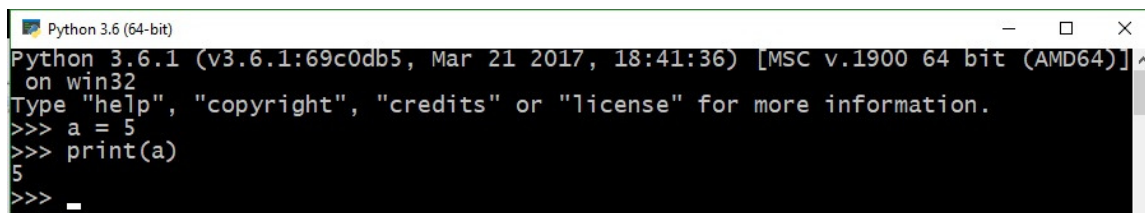
Kapitola 3

Chyby v jazyku Python

Jazyk Python je interpretový programovací jazyk so silnou formou abstrakcie. Zaraďuje sa teda medzi programovacie jazyky ako C++, Perl alebo Java. Silným levelom abstrakcie sa rozumie veľká odlišnosť písaného zdrojového kódu od kódu strojového.

Pre spracovanie programovacích jazykov so silným levelom abstrakcie sa používajú interprety alebo kompilátory. Interpret preloží zdrojový kód a spustí ho. Kompilátor najprv zdrojový kód preloží a vytvorí objektový kód alebo priamo spustiteľný súbor. Následne je možné tento súbor spustiť. V tú chvíľu, kedy je zdrojový kód preložený, je možné program spúšťať opakovane. Na rozdiel od interpretu, ktorý preklad a spúšťanie vykonáva v jednom kroku.

Interpret jazyka Python je možné použiť dvoma spôsobmi. V interaktívnom móde a ako skript. V interaktívnom móde je komunikácia s interpretom okamžitá. Používa sa špeciálny príkazový riadok, kde je možné zadávať príkazy a interpret ihneď zobrazuje výsledky.



Obr. 3.1: Python interpret.

V druhom prípade je možné uložiť zdrojový kód do súboru a následne ho preložiť a spustiť pomocou interpreta. Súbor musí mať príponu .py aby ho interpret vedel identifikovať. Tento program sa potom označuje ako skript.

Vo všeobecnosti sa interpret v interaktívnom móde používa na testovanie veľmi krátkych programov. Všetko dlhšie ako pár riadkov je efektívnejšie použiť ako skript [3].

3.1 Chyby syntaxu

Ako som už spomenul, syntaktická chyba znamená, že kompilácia zdrojového kódu neprebehla úspešne, nakoľko kompilátor nevedel rozpoznať nejakú konštrukciu. Keďže je jazyk Python jazyk interpretový, o nájdenie týchto chýb sa stará interpret.

Počet typov syntaktických chýb, akých sa dá dopustiť, je veľmi veľa. Častými chybami, rovnako ako pri iných programovacích jazykoch, sú napríklad nesprávny počet otváracích a uzavieracích zátvoriek vo výrazoch alebo vynechanie uvodzoviek pri používaní stringov.

Konkrétnejšie chyby, ktoré je možné priradiť zväčša iba jazyku Python, sú chybné napísané kľúčové slová alebo ich nesprávne použitie. Ďalšou chybou je vynechanie dvojbodky na konci podmienky alebo cyklu. Tieto chyby sú veľmi časté u programátorov, ktorí presedlali z jazyka C na Python.

Najčastejšou chybou, s ktorou som sa stretol, je chyba indentácie. Niekedy interpret neoznačí túto chybu ako *IndentationError* ale priamo za *SyntaxError*. Ku chybe indentácie sa vyjadším v samostatnej sekcii 3.3.

3.2 Chyby sémantiky

Keďže je v jazyku Python vykonávaná kompilácia a spustenie programu v jednom kroku, tak je ich detekcia vrámci času takmer rovnaká. Rozlíšiť ich je možné pomocou chybovej správy.

Jazyk Python je silno typovaný jazyk. Od toho faktu sa odvíjajú aj časté chyby hlavne začínajúcich programátorov. Silno typovaný jazyk znamená, že ak sa napríklad v premennej typu *string* nachádzajú iba celé čísla, nemôže sa z neho len tak stať typ *integer*. Je na to potreba explicitnú konverziu. Preto sa častokrát vyskytuje chyba *TypeError*. Programátor nemá implicitne napísaný typ premennej, ako je tomu v jazyku C. Musí si to sledovať sám alebo ho na to upozorní až interpret pri spustení.

Chybám, ktoré sa vyskytnú za behu programu v Pythone taktiež hovoríme výnimky alebo *Exceptions*. Tieto výnimky je možné odchytať pomocou *try* a *except* bloku. Ten pomáha lepšie špecifikovať vyskytnutú chybu a taktiež reakciu programu na ňu. Ďalšou výhodou je možnosť definovať vlastné výnimky a tým pádom aj chybové správy programu. Zoznam vstavaných výnimiek jazyka Python je si možné prezrieť v jeho dokumentácii [4].

3.3 Chyby indentácie

Jednou z najviac odlišných vlastností jazyka Python je jeho indentácia, čiže formátovanie zdrojového kódu a zarovnanie. Princíp spočíva v tom, že sa zdrojový kód rozdeľuje do blokov, ktoré musia mať rovnaké odsadenie. Ako blok si môžeme predstaviť *if else* podmienku.

Ak v jazyku C použijeme na jednom riadku odsadenie pomocou tabulátora a na druhom pomocou medzier, žiadny problém nenastane. Python je v tomto veľmi striktný a vyžaduje buď jedno alebo druhé. Pri miešaní indentácie interpret vyhodí chybu *IndentationError: unindent does not match any outer indentation level*. Práve toto môže byť pre programátorov zbehlých v iných programovacích jazykoch priam až frustrujúce a nepochopiteľné hľadať jednu zatúlanú medzeru. Mnohé novšie textové editory ale poskytujú možnosť prevodu medzier na tabulátory a naopak. Programátori majú radi voľnosť formátovať si kód, ako len oni chcú.

Nevýhodou takejto vlastnosti je teda zvyk a špecifickosť oproti iným jazykom. Výhodou jednotnej indentácie je ale veľmi čisto formátovaný zdrojový kód, ktorý je dobre čitateľný. Netreba teda lúštiť cudzí kód a zvykať si, ako ten daný človek používa zátvorky.

3.4 Nesprávne praktiky

Rovnako ako jazyk Perl, Python je silno idiomatický programovací jazyk. Väčšinou tu existuje zaužívaný spôsob, akým niečo naprogramovať. V nasledujúcich riadkoch popíšem zopár idiómov.

Pre výmenu hodnôt medzi dvoma premennými použijeme v jazyku C pomocnú premennú.

```
temp = a
a = b
b = temp
```

Niektó by použil rovnaký prístup aj v Pythone, no ten ponúka čistejšie a jednoduchšie riešenie.

```
b, a = a, b
```

Silným pomocníkom sú aj negatívne indexy. Tie nám umožňujú indexovať stringy alebo listy odzadu. Zle navrhnutá funkcia na odňatie prípony mena súboru by vyzerala nasledovne.

```
def get_suffix(word):
    word_length = len(word)
    return word[word_length - 2:]
```

Namiesto toho použijeme negatívny index a vyberieme dva posledné znaky.

```
def get_suffix(word):
    return word[-2:]
```

Ďalším známym ale nie čisto Pythonovským idiómom je zrefazovanie porovnávaní v podmienkach. Namiesto znovuprepisovania premennej

```
if x <= y and y <= z:
    return True
```

môžeme použiť skrátenú verziu.

```
if x <= y <= z:
    return True
```

O týchto a o množstve ďalších idiómov hovorí kniha: *Writing Idiomatic Python* [9].

O detekciu aj takýchto idiómov pre jazyk Python sa snaží táto práca. Je potrebné nájsť nevhodné konštrukcie v kóde a poskytnúť doporučenie na zlepšenie. O návrhu a implementácii na riešenie tohoto problému sa dočítate v nasledujúcich kapitolách.

3.4.1 Chyby programového štýlu

Vrámcí štýlu hovoríme o dvoch hlavných kategóriach. Jedná sa o doporučená pre písanie zdrojového kódu (PEP 8) a dokumentačných komentárov (PEP 257).

Doporučenie PEP 8 [12] sa snaží rozviť jednotnú indentáciu a taktiež zjednotiť písanie rôznych konštrukcií jazyka Python. Jedným z doporučení je písanie každého importu na samostatný riadok.

```
import os
import sys
```

Nesprávne by bolo:

```
import os, sys
```

Toto ale neplatí pre nasledujúci prípad, keďže chceme z knižnice *subprocess* importovať iba niektoré funkcie a nie všetko.

```
from subprocess import Popen, PIPE
```

Ďalšie témy, ktorých sa doporučenie dotýka sú:

- Rozloženie kódu - odriadkovanie, medzery, prázdne riadky, dĺžka riadkov
- Použitie apostrofov v stringoch
- Komentáre
- Pomenovania - nevhodné mená, triedy, objekty, metódy, typy, globálne premenné, premenné, konštanty, funkcie

Doporučenie PEP 257 [6] hovorí o pravidlách, ktoré by sa mali dodržiavať pri písaní dokumentačných komentárov, inak nazvaných, docstringov. Docstring sa používa ako komentár s trocha uvoľnenými na zdokumentovanie určitej časti kódu. Na rozdiel od klasických komentárov nie sú docstringy odňaté pri parsovaní zdrojového kódu do stromu ale zanechávajú sa po dobu behu programu. Docstringy majú jednotnú syntax a vďaka tomu je možné jednoduchšie vytvárať generované dokumentácie ku programom. Pri tvorení väčších projektov sú docstringy veľakrát nevyhnutné a medzi skúsenými programátormi sa stali bežnými.

Najčastejšie sa nimi dokumentujú:

- Moduly
- Funkcie
- Triedy
- Metódy

Príklad docstringu pre popis funkcie dekorátora:

```
def limit_calls(max_calls, error_message):  
    """  
    Decorator for limiting calls.  
  
    :param max_calls: max number of calls  
    :param error_message: error message  
    :type max_calls: int  
    :type error_message: string  
    :returns: function  
    :rtype: object  
    """
```

Na detekciu štylistických chýb existujú vytvorené nástroje. Pre detekovanie chýb v docstringoch je možné použiť externú knižnicu¹.

Pre detekciu chýb v rámci doporučení PEP 8 je možné využiť online nástroj² alebo externú knižnicu³.

¹<https://pypi.python.org/pypi/pep257>

²<http://pep8online.com/>

³<https://pypi.python.org/pypi/pep8>

Kapitola 4

Automatizácia výuky v programovaní

Ako som už načrtol v úvode, hlavným dôvodom automatizácie výuky je jej urýchlenie a hlavne odbremenenie vyučujúcich od časovo náročného kontrolovania každého jedného študenta. Kľúčom ku kvalitnej výuke je spätná väzba študentom. Preto je v tejto dobe nemysliteľné, aby dostal študent spätnú väzbu ku zdrojovému kódu v rovnakom čase, ako sa opravujú písomné práce v škole, čiže o týždeň. Efektivita učenia by bola veľmi nízka. Rýchlosť napredovania technológii si vyžaduje aj napredovanie v oblasti výuky. V tom v posledných rokoch výrazne pomohla výpočetná technika. Ktovie, možno sa o pár rokov ani nebude chodiť fyzicky do školy a všetko sa bude diať elektronicky.

Automatické vyhodnocovanie nabera na sile, čím väčší je počet hodnotených študentov. Výuka pomocou informačných technológii má v rámci automatického vyhodnocovania rôzne typy. V nasledujúcich podkapitolách predstavím niekoľko spôsobov, akými sa dá automatizácia výuky v programovaní využiť v praxi.

4.1 E-learning

Jedným z najznámejších spôsobov je E-learning. Je založený na využití informačných technológii so zámerom vzdelávať. Má podobu kurzov alebo testov, ba dokonca až kvízového charakteru. Je možné ho využívať ako hlavný zdroj výuky alebo iba doplnok.

E-learningové kurzy môžu byť otvorené alebo uzavreté. Otvorené kurzy sú zväčša dostupné online pre všetkých záujemcov, kde sa dá často zakúpiť nejaký ďalší, lepší kurz, ktorý výučbu prehĺbuje.

V rámci uzavretého sa dá celý proces zjednodušene popísať tak, že vyučujúci vytvorí kurz, ten následne nahrá na e-learningový portál, kde ho sprístupní študentom. Študenti si tieto kurzy vypracujú. Kurz môže obsahovať aj bodovaný test, kde si študenti ihneď môžu overiť znalosti, ktoré získali. Každý test by mal po jeho vyplnení študentovi povedať, ktoré odpovede boli správne. To je príklad spätnej väzby.

Kurzy môžu byť založené čisto iba na teoretických vedomostiach alebo naopak kontrolujú priamo písanie zdrojového kódu.

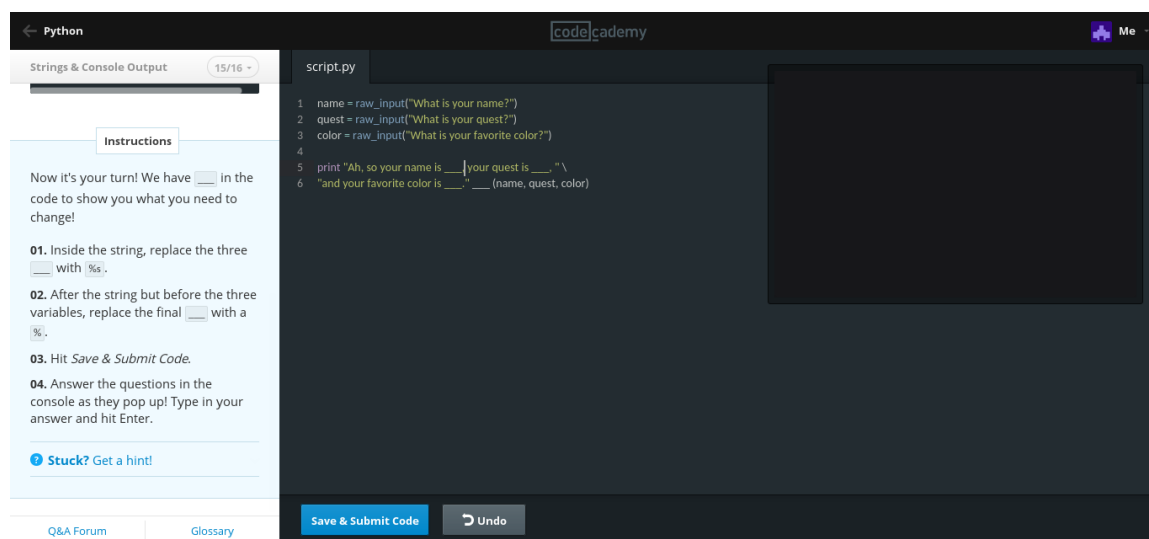
Existuje množstvo voľne dostupných e-learningových kurzov pre začiatočných programátorov. Fungujú na jednoduchom princípe kompilácie kódu a porovnaní oproti správnym hodnotám.

4.1.1 Code School

E-learningový portál, ktorý ponúka široké možnosti výuky programovania. Zameriava sa skôr na skriptovacie jazyky a webový development. Obsahuje voľné, ako aj platené kurzy pre Python, PHP, HTML, CSS a iné. Kurzy sú zamerané skôr na riešenie problémov a analytické zmýšľanie, ako výuku syntaxu jazyka.

4.1.2 Codecademy

Jedná sa asi o najpopulárnejší prostriedok na e-learning z oblasti programovania. Obsahuje kurzy pre najznámejšie skriptovacie programovacie jazyky. Jednoduché kurzy sú voľne dostupné, prepracovanejšie si treba predplatiť. Obsahuje súbory úloh, ktoré postupne učia záujemcov základy syntaxu, ako aj vstupy, výstupy programov a použitie jednotlivých dátových štruktúr. Momentálne obsahuje 10 hodinový voľne dostupný kurz pre jazyk Python. Viac informácií na samotnej stránke.¹



Obr. 4.1: Príklad kurzu v Codecademy.

4.2 Analýza kódu

Pre naše potreby je použiteľnejším príkladom systém, ktorý nám pomôže bližšie pochopiť, ako každý kus kódu vlastne funguje. Analýzou môžeme zistiť informácie, ktoré z bežného čítania zdrojového kódu nezistíme. Tieto informácie nám môžu pomôcť lepšie pochopiť kód, ako aj nájsť chyby alebo zvýšiť jeho efektivitu.

Podobný prístup popisuje aj práca: *Fully Automatic Assessment of Programming Exercises* [13], ktorá využíva e-mail, ako platformu pre prenos súborov od študentov na kontrolu. Spätná väzba je následne poskytnutá taktiež cez e-mail. Systém bol navrhnutý skôr pre menšie projekty programovacieho jazyka *Scheme* a je možné do neho profesormi pridávať nové úlohy. Ohlasy študentov na tento systém boli prevažne kladné.

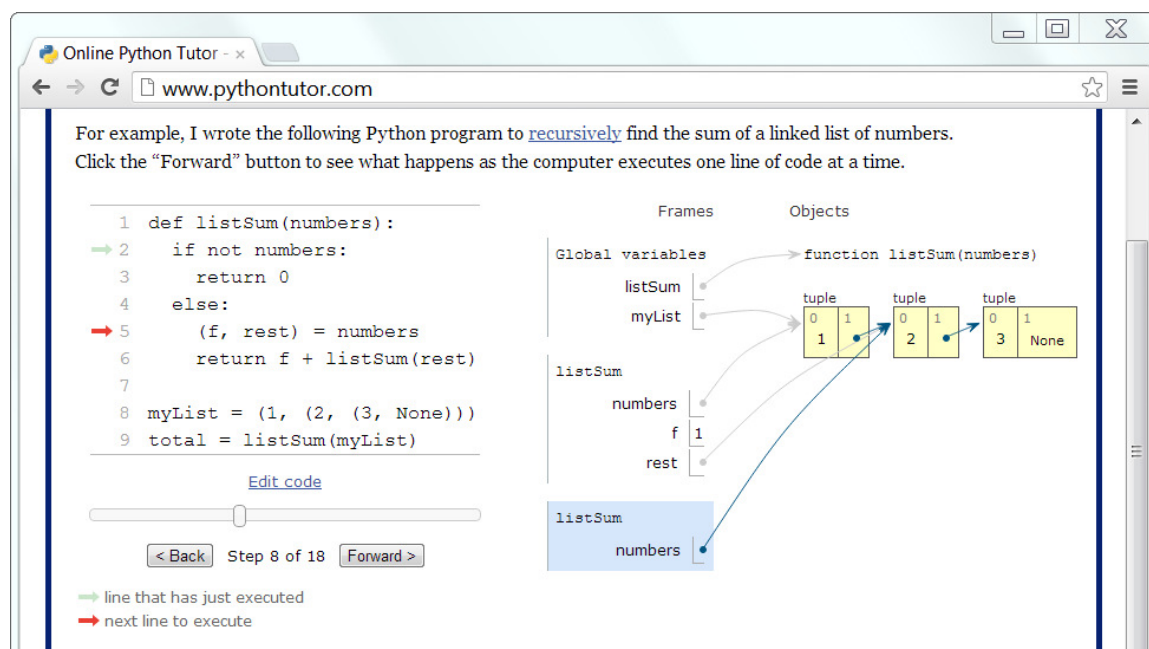
¹<https://www.codecademy.com/>

Existuje množstvo voľne dostupných online interpretov jazyka Python, kde je možné si odskúšať svoj zdrojový kód. Napríklad *repl.it* ², *OnlineGDB* ³ alebo *JDoodle* ⁴. Tie ale neponúkajú nič špeciálne, čo by sme nezískali z nejakého IDE.

4.2.1 Python Tutor

Python Tutor je voľne použiteľná webová aplikácia na vizualizáciu prechodu interpreta cez zdrojový kód. Pri prechode je možné sledovať obsah premenných a ako sa zobrazované príkazy vyhodnocujú. Je to veľmi silný prostriedok na výuku a pochopenie jazyka. Aplikácia okrem jazyka Python podporuje aj jazyky Java, JavaScript, TypeScript, Ruby, C, alebo C++. Jeho hlavnou funkcionalitou je takzvané krokovanie.

Krokovaním sa rozumie postupný prechod kódom po jednotlivých riadkoch. V tom nám pomáha debugger, ktorý umožňuje zastaviť beh programu po každom tomto riadku alebo po predom definovanej zarážke (breakpoint)⁵. Na tomto princípe funguje aj Python Tutor. Viac informácií o ňom na samotnej stránke⁶.



Obr. 4.2: Krokovanie pomocou Python tutora.

4.2.2 Learn Python

Jedná sa o voľne dostupný interaktívny učebný materiál ku jazyku Python ⁷. Témy výuky sú rozdelené podobne, ako v dokumentácii jazyka, čiže od najjednoduchších konštrukcií ako premenné a ich typy až ku dekorátorom. Každá téma obsahuje krátky úvod do problematiky, veľa príkladov a vlastné shelly s predpripraveným kódom na spustenie. Na konci každej témy je cvičenie so správnym riešením, ktoré je možné si prezrieť.

²<https://repl.it/>

³https://www.onlinegdb.com/online_python_compiler

⁴<https://www.jdoodle.com/python-programming-online>

⁵<http://www.fit.vutbr.cz/~martinek/clang/debug.html>

⁶<http://www.pythontutor.com/>

⁷<https://www.learnpython.org/>

Všetky témy sú voľne dostupné a tvorcovia vytvorili rovnaký systém aj pre jazyky Java, HTML, CSS, C, C++, C#, JavaScript, PHP a iné.

Welcome / **Loops**

Get started learning Python with [DataCamp's free Intro to Python tutorial](#). Learn Data Science by completing interactive coding challenges and watching videos by expert instructors. [Start Now!](#)

[◀ Previous Tutorial](#)[Next Tutorial ▶](#)

Loops

There are two types of loops in Python, for and while.

The "for" loop

For loops iterate over a given sequence. Here is an example:

script.py

```
1 primes = [2, 3, 5, 7]
2 for prime in primes:
3     print(prime)
```

IPython Shell

In [1]: |

Run

Powered by DataCamp

Obr. 4.3: Cvičenie ku cyklom jazyka Python.

Kapitola 5

Návrh

Pri návrhu systému bol kladený dôraz na rozdelenie celého procesu vyhodnotenia na jednotlivé časti. Následne bolo možné tieto časti postupne implementovať. Dôraz bol taktiež kladený na to, aby bola všetka funkcionálna skrytá pred užívateľom a použitie bolo intuitívne. Niektoré konkrétnejšie aspekty systému boli riešené až priamo pri jeho implementácii. Táto kapitola popisuje návrh v podobe, v akej bol po naštudovaní príslušnej literatúry a pred implementáciou akejkoľvek časti systému. Celý proces automatického vyhodnotenia projektov a poskytnutia spätnej väzby sa dá popísať nasledujúcimi krokmi v danom poradí.

1. Vstup - príjem súboru so zdrojovým kódom
2. Spracovanie zdrojového kódu a prípravenie na beh testov
3. Spustenie testov na detekciu chýb
4. Prípravenie dát so spätnou väzbou
5. Výstup - zobrazenie spätnej väzby

5.1 Vstup

Celková komunikácia študenta so systémom prebieha pomocou webového rozhrania. To bolo vybrané ako hlavný pilier systému kvôli jednoduchému vývoju.

V dnešnej dobe je web development veľmi rozšírený a existuje množstvo nástrojov, ktoré jeho vývoj uľahčujú. Webové aplikácie sú prístupné z rôznych zariadení. Jediné, čo je potreba, je pripojenie k internetu. Výhodou oproti komunikácii cez e-mail vidím okamžitú spätnú väzbu webového rozhrania [2]. E-mail, ktorý by študent posielal, by musel mať určitý pevný formát. Na druhej strane by bolo potreba zakaždým otvárať novú e-mailovú správu.

Pri webovej aplikácii je taktiež možné prispôbiť si vzhľad podľa vlastnej potreby. Ďalšou výhodou je možnosť prihlásenia študentov a prezerania spätnej väzby ku zdrojovému kódu výhradne pod svojim profilom. Existujú však aj nevýhody ako bezpečnosť a rýchlosť systému v závislosti na vyťažení.

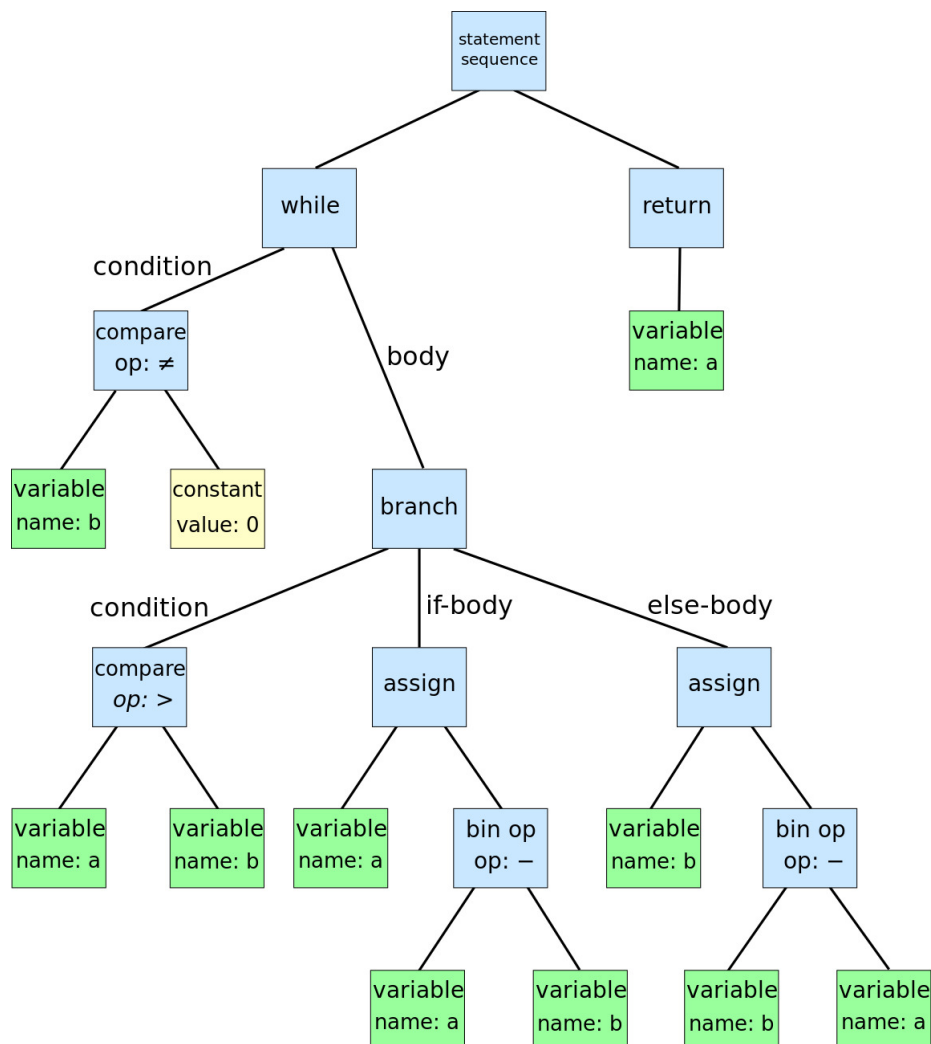
Pomocou webového rozhrania musí študent na titulnej stránke systému zadať svoj login a nahrať súbor so zdrojovým kódom projektu. Pri zadávaní týchto informácií musí splniť niektoré podmienky, ako správny formát loginu a taktiež názvu súboru. Nakoľko sa jedná o systém pre vyhodnocovanie projektov v jazyku Python, budú prijaté jedine súbory s príponou `.py` a žiadne iné. Akonáhle sú tieto podmienky splnené, súbor je prijatý na jeho testovanie.

5.2 Spracovanie zdrojového kódu a pripravenie na beh testov

Po úspešnom prevzatí súboru je potreba kód upraviť na formu, ktorá je efektívna na beh testov. V prvom rade je potrebné zabezpečiť, že je kód možné skompilovať. Ak obsahuje syntaktické chyby, nie je potrebné hľadať vylepšenia, pretože by nesprávne použité štruktúry bránili v ich nájdení.

Bežným prostriedkom pre analýzu zdrojového kódu je abstraktný syntaktický strom. Jedná sa o stromovú reprezentáciu štruktúry zdrojového kódu. Každý list stromu reprezentuje určitú konštrukciu zo zdrojového kódu. Avšak, nie je v ňom reprezentovaná každá konštrukcia. Napríklad zátvorky v *if-else* podmienke sú vynechané, pretože sú implicitné. Týmto sa abstraktný syntaktický strom líši od syntaktického stromu, kde sú tieto konštrukcie zahrnuté. Abstraktné syntaktické stromy sa vo veľkom používajú v kompilátoroch.

Pre jazyk Python existuje knižnica *ast*¹, ktorá slúži na vytváranie a prechádzanie abstraktného syntaktického stromu.



Obr. 5.1: Jednoduchý príklad abstraktného syntaktického stromu.

¹<https://docs.python.org/3/library/ast.html>

Pre bližšie pochopenie a použitie je si možné prečítať rozšírenú dokumentáciu *Green Tree Snakes - the missing Python AST docs* [8].

5.3 Spustenie testov na detekciu chýb

Po úspešnom vytvorení abstraktného syntaktického stromu je možné vykonať testy na detekovanie zlých praktík. Testy sú koncipované ako funkcie. Každý test, čiže každá funkcia, odpovedá hľadaniu jednej praktiky. Ich bližší popis je uvedený v kapitole 6.

V rámci praktík sa testujú použité konštrukcie ako aj ich štýl. Štýl je kontrolovaný podľa doporučení *PEP 8*. Okrem testovania praktík sa testuje aj funkcionálna študentských projektov. Každý zadaný projekt má osobitný súbor testov, kde sa testuje správna funkcionálna vzhľadom ku zadaniu. Bodovanie sa však neuvádza, keďže nie je zámerom systému.

5.4 Pripravenie dát so spätnou väzbou

Skutočnosť, že sa v zdrojovom kóde našli chyby je potreba pretaviť do spätnej väzby. Preto je potreba vytvoriť akýsi záznam, ktorý obsahuje informáciu o chybe, kde nastala, popis zlepšenia a literatúru na dodatočné štúdium. Tento záznam je jednotný pre všetky spúšťané funkcie.

Po prejdení všetkých testov sa tieto záznamy zozbierajú a sformátujú na podobu, ktorá je prijateľná na ďalšie použitie v systéme. Tento záznam predstavuje jadro celkovej spätnej väzby. Keď sú všetky záznamy o chybách pripravené, je možné ich prezentovať študentovi.

5.5 Výstup

Výstupom sa rozumie spätná väzba študentovi. Záznamy o chybách sú formátované pomocou jazyka HTML a predvedené vo webovom rozhraní.

5.6 Webové rozhranie

Cieľom je jeho jednoduchosť a efektivita. Nie sú potrebné žiadne špeciálne funkcie.

Existuje množstvo Python frameworkov na vývoj webových aplikácií. Napríklad *Django*², *web2py*³, *Flask*⁴ alebo *Pyramid*⁵.

Pre vývoj tohoto systému bol vybraný *Flask*. Hlavným dôvodom bol fakt, že *Flask* je takzvaný non full-stack framework, čo znamená, že neobsahuje množstvo funkcií na vývoj veľkých aplikácií ako napríklad *Django*, ktoré by mohli byť zahŕňajúce pri používaní. Jedná sa o minimalistický framework, ktorý obsahuje základné funkcie potrebné na vývoj webových aplikácií. Vyniká svojou jednoduchosťou použitia aj pre neskúsených webových developerov. Obsahuje taktiež aj debug mode, ktorý poskytuje developerovi spätnú väzbu v momente keď sa vyskytne chyba. Server následne pri chybe nie je potrebné zakaždým reštartovať a spúšťať znova ale framework sa o to postará sám. Pre priamočiarosť vyvíjaného systému je dobrou voľbou.

²<https://www.djangoproject.com/>

³<http://www.web2py.com/>

⁴<http://flask.pocoo.org/>

⁵<https://trypyramid.com/>

Pre zobrazenie spätnej väzby je obrazovka delená na dve polovice. Jedna zobrazuje študentov kód a druhá obsahuje zoznam chýb a zlepšení. V rámci štýlu je formát rovnaký. Kód na jednej strane, chyby štýlu na druhej. Obrazovky na spätnú väzbu pre idiómy a štýl sú oddelené kvôli prehľadnosti.

Kapitola 6

Implementácia

Samotná implementácia systému neprebiehala v rovnakom poradí, ako návrh. V prvom rade bolo vytvorené veľmi zjednodušené webové rozhranie. Dôvodom bola možnosť predstaviť si systém ako celok a takisto možnosť vidieť všetky zmeny vykonané v systéme v takpovediac konečnom výsledku. Hovoríme o kostre rozhrania s minimálnou funkcionalitou.

Nasledovalo spracovanie súboru so zdrojovým kódom. Najväčšou časťou implementácie sú testy na vyhľadanie nesprávnych praktík. Tu platilo čím viac testov, tým lepšie. Prednosť avšak zo začiatku dostávali testy na chyby, ktoré je jednoduché v zdrojovom kóde urobiť. Tým bolo možné zvýšiť šancu systému na úspech pri hľadaní chýb. Po určitom množstve vytvorených testov bolo potrebné vytvoriť spätnú väzbu. Vďaka vopred vytvorenému webovému rozhraniu bolo možné vidieť predbežný výsledok a predstaviť si ďalší postup.

Samozrejme, počas celého procesu implementácie bolo potreba každú časť systému upravovať a zlepšovať. Takisto bola časom pridávaná rôzna funkcionalita. Napríklad, ošetrenia chýb pri zadávaní súborov do systému, kontrola štýlu zdrojového kódu alebo pomocné funkcie, ktoré uľahčili vykonávanie zložitejších operácií. Po celú dobu boli taktiež pridávané nové testy. Všetky tieto prvky nebudú popísané chronologicky ale v podkapitolách, ku ktorým patria.

6.1 Webové rozhranie

Hlavný pilier systému sa nachádza v súbore *flask_app.py*. Z tohoto súboru sa spúšťa flask aplikácia. Najprv je ju ale potrebné vytvoriť pomocou príkazu `app = Flask(__name__)` a následne spustiť.

```
if __name__ == '__main__':  
    app.run()
```

Webového rozhranie je možné z pohľadu implementácie rozdeliť na dve časti.

- Routing - riadi a limituje navigáciu študenta po systéme. Definuje konečný počet ciest v URL adrese systému.
- Render - stará sa o grafickú stránku každej sprístupnenej adresy systému. Pomocou neho študent interaguje so systémom.

Systém je nasadený na fakultnom serveri *ivs* na adrese *ivs.fit.vutbr.cz:8090/proj*. Bližšie informácie ohľadom servera sú rozpísané v sekcii 6.7.

Domovskou stránkou je teda URL server s príponou */proj*. Je to titulná stránka systému, ktorú študent uvidí prvú. Obsahuje jednoduché grafické prvky a hlavne pole na zadanie loginu študenta a výber súboru na kontrolu (viď. príloha B.1).

Po úspešnom vyplnení loginu a pripojení súboru sú okamžite spustené všetky testy. Ak sa nevyskytne syntaktická chyba, tak je študent presmerovaný na adresu *ivs.fit.vutbr.cz:8090/proj/<login>*. Tu *<login>* predstavuje login, ktorý študent zadal pri vstupe do systému. Táto stránka reprezentuje sumár výsledkov testovania rozdelených do dvoch častí (viď. príloha B.2).

Praktiky kódu a doporučená PEP 8. Každá časť obsahuje počet nájdených chýb a je si ju možné rozkliknúť pre bližšie informácie. Po rozkliknutí doporučení PEP 8 je študent presmerovaný na adresu *ivs.fit.vutbr.cz:8090/proj/<login>/pep8*. Stránka je rozdelená na 2 polovice. Na ľavej strane je študentský kód a na pravej sú vypísané všetky chyby štýlu (viď. príloha B.3).

Rovnaké rozdelenie platí aj pre *ivs.fit.vutbr.cz:8090/proj/<login>/praktiky*, kde sa avšak na pravej strane nachádzajú vypísané doporučená ku zdrojovému kódu študenta (viď. príloha B.4).

Stránky *.../proj/<login>*, *.../proj/<login>/pep8* a *.../proj/<login>/praktiky* navyše obsahujú tlačidlo *Testovať ďalší súbor*. Po jeho stlačení je študent presmerovaný na domovskú stránku systému a môže kontrolu opakovať pre iný súbor. Taktiež sa pri jeho stlačení mažia dočasné súbory a uvoľňujú užívateľské premenné takzvané *session variables*.

Ku každej stránke patrí vlastný súbor definujúci grafické rozhranie danej stránky. Tieto súbory sú napísané v jazyku *HTML*. Sú tu využité jednoduché konštrukcie a ich štýl je dotvorený jazykom *CSS* v súbore *style.css*.

HTML súbory sú priradené ku stránkam nasledovne:

- *ivs.fit.vutbr.cz:8090/proj* → *home.html*
- *ivs.fit.vutbr.cz:8090/proj/<login>* → *eval.html*
- *ivs.fit.vutbr.cz:8090/proj/<login>/pep8* → *pep8.html*
- *ivs.fit.vutbr.cz:8090/proj/<login>/praktiky* → *praktiky.html*

6.2 Vstup

Zadávané informácie pri vstupe do systému je potreba dostatočne ošetriť, aby sme sa vyhli zbytočným a neočakávaným chybám. Táto kontrola sa delí na kontrolu loginu a kontrolu zadávaného súboru so zdrojovým kódom. Tieto dve kontroly sú ale navzájom blízko späté.

Pri kontrole súboru treba v prvom rade ošetriť či sa jedná o pythonovský súbor. O toto sa postará funkcia `allowed_file(filename)`. Z názvu súboru je vyextrahovaná prípona a tá je následne porovnaná oproti dovoleným príponám, v našom prípade iba *.py*. Povolené prípony je možné meniť v súbore *constants.py* nasledovne:

```
ALLOWED_EXTENSIONS = set(['py'])
```

Následne sa testuje pomenovanie súboru. Podľa zadání projektov z predmetu ISJ musí mať súbor pomenovanie podľa formátu *isj_projXX_xlogin00.py*, kde *XX* odpovedá poradovému číslu projektu a *xlogin00* loginu študenta. Niektoré loginy avšak nemusia obsahovať

za menom študenta čisto čísla ale sú prípady, kde obsahujú napríklad 1c alebo 1b. Toto bolo taktiež zohľadnené.

Ďalšou podmienkou je, aby časť xlogin00 z názvu súboru odpovedala loginu zadanému v samostatnom poli. Tieto dva loginy sa potom už len jednoducho porovnávajú.

Ošetrené sú samozrejme aj prípady, kedy by bolo pole loginu prázdne alebo by nebol priložený žiadny súbor.

Ak sú všetky tieto podmienky splnené, tak je súbor uložený v zložke */projects* a pripravený na ďalšie spracovanie a beh testov. V prípade akejkoľvek chyby je na titulnej stránke zobrazená chybová správa.

6.3 Spracovanie zdrojového kódu

Aj keď nie je kontrola syntaxu hlavou témou tejto práce, je potrebné ju urobiť ešte pred spustením testov. Kontrolu vykonáva funkcia `syntax_check(filename)`. Súbor je spustený a v prípade syntaktickej chyby je zobrazená chybová správa z interpreta jazyka Python. Táto správa je zobrazená už na titulnej stránke systému. Dôvodom je študentovi ukázať, že nespustiteľné skripty systém neprijíma a je potrebné všetky syntaktické chyby vopred opraviť. V prípade chyby je uložený súbor so zdrojovým kódom vymazaný z priečinka */projects*.

Po vyskúšaní práce s knižnicou *ast* som sa rozhodol uľahčiť si prístup a pracovať s XML formou zdrojového kódu. Výhoda práce s XML štruktúrou je ľahšia čitateľnosť dát a navigácia medzi uzlami. Štruktúra abstraktného syntaktického stromu je prekonvertovaná na jej XML reprezentáciu. Celá konverzia sa odohráva v súbore *ast2xml.py* v nasledovnom poradí:

- Funkcia `convert(filepath)` načíta odovzdaný súbor a pomocou knižnice *ast* je vytvorený abstraktný syntaktický strom.
- Štruktúra stromu je predaná triede `ast2xml(ast.NodeVisitor)`, ktorá preparsuje daný strom do XML podoby. Táto podoba je vytvorená pomocou knižnice *lxml* ¹ modulom *etree*. Bližší popis knižnice je možné nájsť tu: [14].
- Posledným krokom konverzie je upraviť XML štruktúru do podoby lepšie čitateľnej pre programátora. Myslí sa tým odsadenie elementov na základe ich vnútornej štruktúry. To je zabezpečené funkciou `toprettyxml()` z knižnice *xml.dom.minidom* ².

¹<http://lxml.de/>

²<https://docs.python.org/3.6/library/xml.dom.minidom.html>

Výsledná XML štruktúra jednoduchého programu môže vyzeráť napríklad takto:

```
<?xml version="1.0" ?>
<ast>
  <Module>
    <body>
      <Assign col_offset='0' lineno='3'>
        <targets>
          <Name col_offset='0' id='a' lineno='3'>
            <Store/>
          </Name>
        </targets>
        <Num col_offset='4' lineno='3' n='5' />
      </Assign>
      <If col_offset='0' lineno='4'>
        <Compare col_offset='3' lineno='4'>
          <Name col_offset='3' id='a' lineno='4'>
            <Load/>
          </Name>
          <ops>
            <Eq/>
          </ops>
          <comparators>
            <Num col_offset='8' lineno='4' n='5' />
          </comparators>
        </Compare>
        <body>
          <Return col_offset='4' lineno='5'>
            <NameConstant col_offset='11' lineno='5' value='True' />
          </Return>
        </body>
      </If>
      <orelse>
        <Return col_offset='4' lineno='7'>
          <NameConstant col_offset='11' lineno='7' value='False' />
        </Return>
      </orelse>
    </body>
  </Module>
</ast>
```

Týmto je konverzia hotová a pre navigáciu v XML štruktúre sa používa dotazovací jazyk *xpath*. Ten je v navigácii cez uzly veľmi efektívny a jeho použitie je priamočiare. V rámci tohoto projektu je použitá verzia dotazovania podľa definície knižnice lxml³. Príklad dotazu v jazyku *xpath* je bližšie popísaný v sekcii 6.4.1.

³<http://lxml.de/xpathxslt.html>

6.4 Testy na detekciu chýb

Testy sú podľa typu rozdelené na 4 menšie skupiny.

1. Testy idiómov
2. Testy projektov
3. Testy štýlu
4. Iné testy

6.4.1 Testy idiómov

Všetky testy nesprávnych praktík sú umiestnené v súbore *checks.py*. Vyhľadávanie nesprávnych praktík je rozdelené do funkcií, kde každá funkcia predstavuje vyhľadanie konkrétnej praktiky. Tieto testy sú vykonávané na všetkých zaslaných projektoch bez ohľadu na poradové číslo projektu 6.4.2. Vykonávajú sa na vopred vytvorenej XML reprezentácii abstraktného syntaktického stromu. Dotazovacím jazykom *xpath* je možné vyhľadať, čo je potrebné pre nájdenie chyby.

Proces vyhľadávania predvediem na jednoduchom príklade, kde sa snažíme nájsť neidiomatické použitia otvárania súboru pomocou funkcie `open()`. Správnu praktikou by malo byť použitie štýlom `with open()`. Samozrejme sa môžu vyskytnúť výnimky, kedy toto pravidlo neplatí, no vo všeobecnosti je toto lepší prístup.

```
.  
.   
.   
<body>  
  <Expr col_offset='0' lineno='2'>  
    <Call col_offset='0' lineno='2'>  
      <Name col_offset='0' id='open' lineno='2'>  
        <Load/>  
      </Name>  
      <args>  
        <Name col_offset='5' id='path_to_file' lineno='2'>  
          <Load/>  
        </Name>  
        <Str col_offset='19' lineno='2' s='r' />  
      </args>  
      <keywords/>  
    </Call>  
  </Expr>  
</body>  
.   
.   
.
```


Takto vyzerá XML reprezentácia príkazu `open(path_to_file, 'r')`. Na základe týchto dát vieme vytvoriť dotaz *xpath*, ktorý túto konštrukciu vyhľadá. Tento dotaz vyhľadá v zdrojovom kóde študenta všetky výskyty volania funkcie `open()`.

```
...xpath('..//Call/Name[@id="open"]')
```

Návratovou hodnotou funkcie *xpath* môže byť napríklad názov elementu, atribút, hodnota atribútu alebo dokonca samotný element, ako je tomu vo vyššie uvedenom príklade. Ak je teda dotazom praktika nájdená, vytvorí sa záznam o chybe. Formát a príklad záznamu je rozpísaný v sekcii 6.5.

Pre porovnanie uvediem XML štruktúru volania funkcie `with open(path_to_file, 'r')`.

```
.
.
.
<body>
  <With col_offset='0' lineno='2'>
    <items>
      <withitem optional_vars='None'>
        <Call col_offset='5' lineno='2'>
          <Name col_offset='5' id='open' lineno='2'>
            <Load/>
          </Name>
          <args>
            <Name col_offset='10' id='path_to_file' lineno='2'>
              <Load/>
            </Name>
            <Str col_offset='24' lineno='2' s='r' />
          </args>
          <keywords/>
        </Call>
      </withitem>
    </items>
  </body>
  <Continue col_offset='1' lineno='3' />
</body>
.
.
.
```

Rovnaký prístup je použitý vo všetkých funkciách na vyhľadanie nesprávnych praktík. Tie sú už komplikovanejšie oproti zobrazenému príkladu a u niektorých sú použité pomocné funkcie, ktoré sú definované v súbore *helper.py*.

Existujú však aj prípady, kedy nie je potreba pre nájdenie nesprávne použitej konštrukcie vygenerovaná XML štruktúra. Jedná sa o prípady pamäťovej náročnosti pri riešení niektorých problémov. Nasledujúca funkcia vracia prvý neopakujúci sa znak z reťazca.

```
def function(string):
    for i in string:
        if string.count(i) == 1:
            return i
```

Zložitosť tohoto riešenia je pri najhoršom prípade $O(n^2)$. Na druhej strane, keby použijeme špeciálnu funkciu `Counter()`, tak sa časová zložitosť zníži na $O(n)$.

Námet na nesprávne praktiky vyhľadávané v rámci tohoto systému sú prevažne čerpané z knihy *Writing Idiomatic Python* [9].

6.4.2 Testy projektov

V rámci predmetu ISJ bolo tento rok zadaných osem projektov. Prvé tri projekty mali predpísanú kostru, do ktorej mali študenti dopĺňať iba časti kódu. Kostry je možné nájsť v priečinku */zadania*. 4. až 7. projekt bolo potreba napísať celý. 8. projekt bol jediný, ktorý obsahoval viac ako jeden súbor na odovzdanie a ich pomenovanie nezapadalo do predošlej koncepcie projektov. Z toho dôvodu som ku tomuto projektu žiadne testy nevytvoril. U všetkých projektoch je na testovanie použitý dotazovací jazyk *xpath* a chyby sú zobrazené, iba ak študent poskytol systému nejaké riešenie. To by malo odradiť špekulantov, ktorí do systému pošlú prázdne súbory a očakávajú nápoedu.

Konkrétny popis projektov z predmetu Skriptovacie jazyky, ako aj výsledky testovania týchto projektov je možné nájsť v kapitole *Testovanie* 7.

6.4.3 Testy štýlu

Implementáciu štylistických testov je možné rozdeliť na dve skupiny:

- Testy doporučení PEP 8
- Test výskytu dokumentačných komentárov - *docstrings*

Kontrola doporučení PEP 8 sa vykonáva vo funkcii `pep8_check(filepath)`. Samotný test je možný vďaka knižnici *pycodestyle* ⁴. Výstup je uložený do textového súboru. Následne je obsah tohoto súboru načítaný funkciou `pep8_make_html(filename)` a tá pomocou balíčka *pepper8* ⁵ vytvorí pekne formátovanú HTML reprezentáciu týchto doporučení, ktorú je už potom jednoduché zobrazit vo webovom rozhraní.

Podľa zadání projektov z ISJ bolo po študentoch žiadané písať dokumentačné komentáre (ďalej už iba *docstringy*) od štvrtého projektu. Ich výskyt som testoval pri všetkých definovaných funkciách, triedach a metódach. Nebol teda kontrolovaný ich formát ale iba výskyt. *Docstringy* je vhodné písať pri každej konštrukcii, ktorá potrebuje objasnenie pre čitateľa, no táto definícia je príliš obsírna na to, aby boli takéto komentáre predmetom testovania.

⁴<http://pycodestyle.pycqa.org/en/latest/intro.html>

⁵<https://pypi.org/project/pepper8/1.0.2/>

6.4.4 Iné testy

Sem zaradujem testy, ktoré majú takpovediac vlastnú kategóriu.

Shebang ⁶ je široko využívaný v unixových systémoch. Jeho funkciou je označiť daný súbor za spustiteľný. To umožňuje shebang samostný. Jedná sa vlastne o špeciálnu postupnosť znakov, ktorá je situovaná na prvom riadku súboru. Táto postupnosť znakov špecifikuje interpret, ktorý sa má postarať o beh skriptu. V praxi to znamená, že nie je potreba spúšťať súbor ako `python file.py` ale stačí `./file.py`.

Shebang je v rámci projektov kontrolovaný v každom z nich. Stará sa o to funkcia `shebang_check(filepath)`, ktorá porovnáva reťazec prvého riadku súboru oproti povoleným shebangom zo súboru konštánt `/constants.py`.

```
SHEBANG = ("#!/usr/bin/env python", "#!/usr/bin/env python3")
```

Ďalším testom je test na plagiátorstvo. Tento test je veľmi zjednodušený a ani zďaleka nemá takú výkonnosť ako známy prostriedok *MOSS* ⁷, nakoľko sa na tento aspekt práca nezameriava. Ide o porovnanie každého elementu z vopred vytvorených XML štruktúr. Buď sa zistí úplná zhoda alebo nič, avšak kontrola neberie ohľad na názvy premenných ani na čísla riadkov. Konštrukcie teda musia byť v rovnakom poradí. Funkcia `plagiatism(e1, e2)` porovnáva tieto dve XML štruktúry. Pri nájdení zhody je vytvorený textový súbor *PLAGIAT-isj_projXX_xlogin00.py.txt*, ktorý obsahuje názvy súborov, u ktorých sa zhoda našla. Súbor je vytvorený v koreni adresára systému.

6.5 Záznam o chybe

Tento záznam sa vytvára pri testoch idiómov 6.4.1, testoch projektov 6.4.2 a testoch na výskyt docstringov 6.4.3. Formát tohoto záznamu je *list*:

```
['POPIS CHYBY + DOPORUCENIE + LITERATURA', CISLO RIADKA]
```

Čísliel riadkov môže byť aj viac a sú oddelené čiarkami a literatúru obsahuje každý záznam. Všetky hodnoty majú typ *string*. Popis chyby, doporučenie a literatúra sú chápané ako jeden ucelený text. Tento text je predom predpísaný a nachádza sa v súbore `/constants.py`. Tieto doporučenia sú tu definované pre všetky chyby, ktoré systém hľadá.

Volania všetkých funkcií na kontrolu idiómov a chýb sú zastrešené vo funkcii `checks(xml, filepath, filename)` v súbore `checks.py`. Tu sú volané a zozbierané všetky záznamy z testov idiómov ale testy projektov sú volané iba pre dané poradové číslo projektu. Návratová hodnota funkcie je teda *list* všetkých záznamov.

```
[ZAZNAM1, ZAZNAM2, ... ,ZAZNAMX]
```

6.5.1 Logovanie

Z dôvodu archivovania výsledkov testovania zdrojových kódov pre štatistické alebo iné účely je vytvorený jednoduchý logovací systém. Log je vytvorený z každého výsledného listu záznamov, ktorý nie je prázdny. Pre jedinečnosť názvu súboru z dôvodu ich veľkého počtu je použitá funkcia `uuid4()` z knižnice *uuid* ⁸ na vygenerovanie jeho názvu.

⁶[https://en.wikipedia.org/wiki/Shebang_\(Unix\)](https://en.wikipedia.org/wiki/Shebang_(Unix))

⁷<https://github.com/soachishti/moss.py>

⁸<https://docs.python.org/3/library/uuid.html>

6.6 Zobrazenie spätnej väzby

Výsledný list záznamov je predaný HTML súboru *praktiky.html*, ktorý zobrazuje zozbierané chyby vo svojej tabuľke. Pre komunikáciu medzi premennými jazyka Python je v rámci frameworku flask použitý jazyk *Jinja2* [11]. Vďaka nemu je možné vo vnútri HTML konštrukcii používať podmienky alebo cykly. Zobrazený úsek kódu slúži na prechod cez list záznamov a ich výpis.

```
<div class='container-popis'>
  <table style='width:100%'>
    <tr>
      <th>Line</th>
      <th>Popis chyby</th>
    </tr>
    {% for item in result %}
      {% if item %}
        <tr>
          <td>{% for i in item[1:] %}{{i}}, {% endfor %}</td>
          <td>{{item[0]}}</td>
        </tr>
      {% endif %}
    {% endfor %}
  </table>
</div>
```

6.7 Deployment

Nasadenie systému do prevádzky prebehlo až úplne nakoniec. Pre nasadenie bol využitý webový server *Twisted Web* ⁹ s pripojeným *WSGI* kontajnerom ¹⁰. Systém beží na porte 8090. Táto hodnota bola dohodnutá a je ju možné v prípade potreby zmeniť priamo v súbore *flask_app.py*.

Pre zachovanie vnútornej integrity servera *ivs* je použitá voľne dostupná distribúcia jazyka Python s názvom *Anaconda* ¹¹. V rámci tohoto projektu je využitá jej funkcia vytvorenia prostredia. Toto prostredie dovoľuje inštaláciu všetkých potrebných závislostí, ktoré sú potrebné na beh systému.

Pre zabezpečenie, aby systém bežal nepretržite, je použitý linuxový príkaz *screen* ¹². Ten umožňuje spustiť systém v osamotenom procese a následne sa od neho odpojiť bez jeho vypnutia. Navyše umožňuje aj znova pripojenie k bežiacemu procesu.

Pre potreby reštartovania alebo vypnutia systému uvediem postup príkazov. Ak systém beží, je potrebné na serveri *ivs* použiť príkaz **screen -ls**. Ten vypíše všetky bežiace procesy. Následným **screen -X -S ID quit**, kde ID označuje číslo procesu, vypneme daný proces.

⁹<http://flask.pocoo.org/docs/0.12/deploying/wsgi-standalone/#twisted-web>

¹⁰<https://twistedmatrix.com/documents/15.2.1/web/howto/web-in-60/wsgi.html>

¹¹<https://conda.io/docs/index.html>

¹²http://www.linuxsoft.cz/article.php?id_article=1100

Príkazom `screen` vytvoríme nový proces. Spustíme vopred vytvorené prostredie *Anaconda* príkazom `source activate bc`. Následne navigujeme do priečinka aplikácie: `cd /mnt/data/isj-2017-18/public/app`. Ostáva už iba spustiť systém.

```
PYTHONPATH=. twistd -n web --wsgi flask_app.app
```

V rámci *screen* je sa potom možné pomocou *CTRL+A* nasledovaným *CTRL+D* odpojiť od procesu.

Kapitola 7

Testovanie

Funkcionalitu systému je potreba overiť. Už len z dôvodu vývoja a vylepšovania výsledného produktu je táto fáza veľmi dôležitá. Testovanie rôznych častí programu počas jeho vývoja je dôležitým aspektom zaručujúcim správne fungovanie vyvíjaného systému [16]. No pre webové aplikácie je navyše testovanie v praxi neodmysliteľným prostriedkom na získavanie spätnej väzby ohľadom produktu. V tejto kapitole popíšem výsledky testov, ktoré boli vykonané na vyvinutom systéme ako aj problémy, ktoré sa vyskytli.

V rámci tejto bakalárskej práce bol systém nasadený do prevádzky ako podporná výuková pomôcka predmetu *ISJ* v akademickom roku 2017/2018. Z dôvodu nedokončeného vývoja a iných súvislostí bol avšak v aktívnom režime iba pre 6. a 7. projekt. Tým sa rozumie, že testovanie prebiehalo súbežne s termínom odovzdávania projektov podľa termínov v predmete *ISJ*. Študenti si na adrese systému mohli otestovať svoje riešenia a overiť správnosť pythonovských konštrukcií ešte pred samotným odovzdaním projektu. Systém bol ale dodatočne testovaný na všetkých odovzdaných skriptoch pre 1. až 5. projekt.

Celkový počet študentských prác pre 1. až 7. projekt, z ktorých sú tu zozbierané dáta prezentované, je **1063**. U každého projektu v krátkosti popíšem informácie jedinečné ku danému projektu, výskyt chýb ako aj ich počet.

7.1 1. projekt

Prvý projekt bol zameraný na použitie regulárnych výrazov. V ňom som sa snažil študentov popohnať k použitiu nasledujúcich konštrukcií v závislosti na vhodnosti použitia ¹.

- Positive lookahead - `?=`
- Negative lookahead - `?!`
- Positive lookbehind - `?<=`
- Negative lookbehind - `?<!`

Kostra projektu obsahovala aj vlastné testy funkcionality v podobe *assertions*. U nich bolo potreba skontrolovať, že ich študenti nevymažú alebo jednoducho neupravajú podľa svojej potreby, aby im testy prešli. Všetky testy sú definované formou funkcií v súbore *checks_proj01.py*.

¹<https://www.regular-expressions.info/lookaround.html>

V rámci 1. projektu bolo otestovaných **193** študentských prác. Kostra zadania sa nachádza v `/zadania/isj_proj01_xnovak00.py`.

- **90** študentov dostalo doporučenie pre funkciu `first_task(animals)` na použitie `negative` a `positive lookahead`. Funkcia bola perfektná na ich použitie a doporučenie sa zobrazilo, ak chýbalo použitie čo i len jednej metódy.
- **84** študentov dostalo doporučené pre funkciu `second_task(condensed)`. Podobne ako pri prvej funkcii, jednalo sa o `negative lookahead`, `positive lookahead` ako aj `positive lookbehind`.
- **12** študentov nevypracovalo zadania správne a neprešli im priložené asserty, na čo boli následne upozornení.
- **10** testovaných prác skončilo na titulnej stránke systému s chybou syntaxu.
- **1** študentovi bolo odporučené použiť `return` na návrat operácii, ako len na návrat premennej.

7.2 2. projekt

Druhý projekt sa sústreďoval na prácu so štruktúrami ako *list*, *set* a podobne. Napríklad vo funkcii `first_task()` bolo potreba z listu vymazať duplikáty. Najlepším riešením bolo zmeniť *list* na *set*. Z definície funkcie *set* vypláva, že sa jedná o štruktúru unikátnych elementov².

Taktiež obsahoval cvičenia na prácu so stringami a ich indexovaním. Tu som kontroloval použitie metódy *string slicing*, ktorá je veľmi efektívna pri práci so stringami³.

Tento projekt taktiež obsahoval vlastné testy funkcii, no tentokrát v podobe výpisu cez `print()`. Tým pádom museli byť ošetrené všetky integrované testy proti prepísaniu rovnako, ako všetky testované funkcie. Všetky testy sú definované formou funkcii v súbore `checks_proj02.py`.

V rámci 2. projektu bolo otestovaných **194** študentských prác. Kostra zadania sa nachádza v `/zadania/isj_proj02_xnovak00.py`.

- **72** študentov dostalo doporučenie na použitie funkcie `set()` s operáciou *intersection* vo funkcii `second_task()`.
- **38** študentov dostalo v rámci funkcie `third_task()` doporučenie na použitie funkcie `counter()`. Indikáciou mohol byť `import collections` navrchu zadania kostry projektu.
- **14** študentov nepoužilo v `first_task()` funkciu `set()` na mazanie duplikátov z listu, ktorá bola na toto cvičenie doslova žiadaná.
- **6** študentov sa pokúšalo v rámci `first_task()` počítat prvky listu inak ako pomocou `len()` a boli na to upozornení.
- **3** práce nevyhovovali vstavaným testom na funkcionálnosť.

²<https://docs.python.org/2/library/sets.html>

³<https://developers.google.com/edu/python/strings>

- 1 práca mala chybný alebo žiadny shebang.
- 1 študent dostal doporučenie na použitie slicingu pri práci so stringami.
- 1 študent dostal doporučenie na použitie `join()` pri spájaní stringov vo funkcii `fifth_task()`.
- 1 práca obsahovala syntaktickú chybu.

7.3 3. projekt

Tretí projekt sa zameriaval na použitie kombinácii funkcii nad štruktúrami. Vyžadovalo sa napríklad použitie *dictionary comprehension* ⁴. Jedná sa o šetrný zápis komplikovanejších konštrukcií. Z vlastných skúseností viem, že študenti sa takémuto zápisu radi vyhýbajú, pretože nie je dobre pochopiteľný na prvý pohľad.

```
{s: p for s, p in zip(singular, plural)}
```

Projekt obsahoval opäť vlastné testy v podobe *assertions*. Ako v predchádzajúcich projektoch, bolo potreba ošetriť, aby neboli prepísané testy ani žiadne iné konštrukcie z pôvodnej kostry projektu. Všetky testy sú definované formou funkcii v súbore `checks_proj03.py`.

V rámci 3. projektu bolo otestovaných **187** študentských prác. Kostra zadania sa nachádza v `/zadania/isj_proj03_xnovak00.py`.

- **187** študentov čiže všetci dostali na doporučenie použiť `return` na návrat operácii, ako len na návrat premennej. Toto vysoké číslo je z dôvodu koncepcie kostry zadania, ktorú pochopiteľne študenti nemali. Doporučenie je teda neopodstatnené.
- **34** študentov dostalo doporučenie vo funkcii `match_permutations(string, words)` na použitie funkcie `sorted()`.
- **33** študentov nepoužilo vo funkcii `plur2sing(singular, plural)` funkciu `zip()` aj keď to vyplývalo zo zadania projektu.
- **23** študentov nevypracovalo projekt správne a neprešli im niektoré priložené asserty.
- **3** práce obsahovali syntaktické chyby.

7.4 4. projekt

Štvrtý projekt už bolo potreba vytvoriť celý. Zadaním bolo vytvoriť tri funkcie.

- `can_be_a_set_member_or_frozenset(item)` - zistí či môže byť `item` prvkom množiny.
- `all_subsets(items)` - z `items` vytvorí zoznam odpovedajúci množine všetkých podmnožín.
- `all_subsets_excl_empty(*items, exclude_empty=True)` - rovnako ako `all_subsets(items)`, no prvky môžu byť zaslané funkcii ako samotné argumenty a je si možné pomocou `exclude_empty` vybrať, či bude prázdna množina súčasťou výslednej množiny.

⁴<https://www.python.org/dev/peps/pep-0274/>

Bolo teda potreba ošetriť správne pomenovanie funkcií alebo použitie parametrov. Keďže je potreba projekt napísať celý, nemá integrované testy. Preto súbor *checks_proj04.py* obsahuje ešte aj testy funkcionality. Študentský súbor je importovaný a sú tu testované správne aj chybové vstupy a ich výsledky sú vyhodnocované.

V rámci 4. projektu bolo otestovaných **146** študentských prác. Toto zadanie projektu už neobsahovalo kosťu a taktiež boli vyžadované docstringy.

- **60** študentov dostalo doporučenie na využívanie plného potenciálu `try-except` bloku. Mnohí ho využívali len ako pokus či daná konštrukcia prejde.
- **58** študentov používalo porovnanie premenných ku *boolean* hodnotám. Bolo im doporučené porovnávať tieto premenné priamo: `if variable:` alebo `if not variable:`.
- **46** prác obsahovalo chybné alebo žiadne docstringy.
- **43** študentov dostalo doporučenie na úpravu funkcie `all_subsets_excl_empty()`, nakoľko jeden alebo viac vytvorených testov na jej funkcionality neprešli. Konkrétne testy boli zobrazené.
- **38** študentov dostalo doporučenie použiť `return` na návrat operácii, ako len na návrat premennej.
- **35** študentov bolo upozornených nepoužívať funkciu `isinstance()` v rámci funkcie `can_be_a_set_member_of_frozenset()`, nakoľko nemusi vracat požadované hodnoty. Literatúra k problematike bola dodaná.
- **32** študentov dostalo doporučenie na úpravu funkcie `all_subsets()`, nakoľko jeden alebo viac vytvorených testov na jej funkcionality neprešli. Konkrétne testy boli zobrazené.
- **22** študentov dostalo doporučenie na úpravu funkcie `can_be_a_set_member_of_frozenset()`, nakoľko jeden alebo viac vytvorených testov na jej funkcionality neprešli. Konkrétne testy boli zobrazené.
- **3** študenti dostali doporučenie na použitie `if item in my_list:`, ak chceli porovnávať rovnakú premennú oproti väčšiemu počtu hodnôt.
- **3** práce obsahovali žiadny alebo chybný shebang.
- **1** študent bol upozornený na vodopádovité použitie `if` podmienok.
- **1** študent dostal doporučenie na iterovanie cez štruktúru pomocou `for...in...`

7.5 5. projekt

Piaty projekt bol dosiaľ najkomplexnejší. Bolo potreba implementovať triedu *Polynomial*, ktorá pracuje s polynómami. Tie bude možné vypisovať v matematickej podobe, porovnávať, sčítavať, umocňovať nezápornými celými číslami, derivovať alebo vyčísliť. Polynóm $2x^3 - 3x + 1$ bude možné v rámci triedy instanciovať troma spôsobmi:

- `Polynomial([1,-3,0,2])`

- `Polynomial(1,-3,0,2)`
- `Polynomial(x0=1,x3=2,x1=-3)`

Podľa zadania bolo podmienkou, aby trieda obsahovala metódy `derivative()` a `at_value()`. Ku zadaniu boli priložené *assertions* testy na overenie funkcionality triedy. Tie som ale v súbore `checks_proj05.py` opätovne priložil a rozšíril.

V rámci 5. projektu bolo otestovaných **123** študentských prác. Od tohoto projektu bolo potrebné kontrolovať aj plagiáty. Tie sa kontrolovali aj oproti minuloročným odovzdaným projektom. V 5. projekte systém zhodu nenašiel.

- **46** prác obsahovalo chybné alebo žiadne docstringy u niektorých tried alebo metód.
- **38** študentov používalo porovnanie premenných ku *boolean* hodnotám. Bolo im doporučené porovnávať priamo: `if variable:` alebo `if not variable:`.
- **37** študentov dostalo doporučenie použiť `return` na návrat operácii, ako len na návrat premennej.
- **36** študentov dostalo doporučenie na použitie `isinstance()` pri zisťovaní typu premennej v objektovo orientovanom programovaní. V 5. projekte ide o vytvorenie triedy a jednoduché `type()` nevracia pri použití podtriedy správny výsledok.
- **27** študentov bolo upozornených na vodopádovité použitie `if` podmienok.
- **19** študentov dostalo upozornenie na chybnú funkcionality riešenia na základe `assert` testov. Konkrétne chyby boli zobrazené.
- **12** študentov dostalo doporučenie na využívanie plného potenciálu `try-except` bloku. Mnohí ho využívali len ako pokus či daná konštrukcia prejde.
- **6** prác nemalo implementovanú triedu `Polynomial()`, funkciu `derivative()` alebo funkciu `at_value()`.
- **6** prác malo chybný alebo žiadny shebang.
- **6** študentov dostalo doporučenie na použitie reťazenia funkcií pri práci so stringami. `foo.strip().upper().lower()`.
- **4** študenti nevyužili pythonovskú metódu `(foo, bar) = (bar, foo)` pri výmene hodnôt dvoch premenných ale naopak použili klasickú pomocnú premennú, známu pre jazyk C.
- **3** práce obsahovali syntaktickú chybu.
- **3** práce dostali doporučenie na iterovanie cez štruktúru pomocou `for...in...`
- **1** študent dostal doporučenie na použitie konštrukcie `.join()` pri spájaní stringov nachádzajúcich sa v liste.

7.6 6. projekt

Šiesty projekt bolo taktiež potrebné napísať celý. Jednalo sa o dve funkcie:

- `first_nonrepeating(string)` - vráti prvý neopakujúci sa znak z reťazca `string`.
- `combine4(list, result)` - `list` predstavuje štyri celé kladné čísla a `result` očakávaný výsledok. Cieľom je, aby funkcia vrátila zotriedený zoznam neopakujúcich sa výrazov, poskladaných zo štyroch zadaných čísel a operácii `+`, `-`, `*`, `/` tak, aby výsledok výrazu odpovedal očakávanému výsledku `result`.

Kontroluje sa správne použitie argumentov pri definícii funkcií. Súbor `checks_proj06.py` taktiež obsahuje množstvo testov oboch funkcií s rôznymi vstupmi. Testujú sa aj nesprávne vstupy a sleduje sa, ako na ne študentský skript reaguje.

V rámci 6. projektu bolo otestovaných **177** študentských prác. Tento projekt už bol testovaný v praxi.

V 6. projekte systém našiel dva plagiáty. Na podozrelé práce bolo poukázané garantovi predmetu *ISJ*.

- **84** študentov dostalo upozornenie na nefunkčnosť funkcie `combine4()` v niektorých krajných prípadoch. Avšak niektoré testy testovali až príliš okrajové situácie, ktoré študenti neriešili, a preto je toto číslo vysoké.
- **45** študentov dostalo doporučenie použiť `return` na návrat operácii, ako len na návrat premennej.
- **30** študentov dostalo doporučenie na využívanie plného potenciálu `try-except` bloku. Mnohí ho využívali len ako pokus či daná konštrukcia prejde.
- **27** študentov dostalo upozornenie na nefunkčnosť funkcie `first_nonrepeating()` v niektorých krajných prípadoch.
- **25** prác obsahovalo chybné alebo žiadne docstringy.
- **10** študentov používalo porovnanie premenných ku *boolean* hodnotám. Bolo im doporučené porovnávať tieto premenná priamo: `if variable:` alebo `if not variable:`.
- **8** prác neobsahovalo jednu z dvojice požadovaných funkcií `first_nonrepeating()`, `combine4()`.
- **6** študentov bolo upozornených na vodopádovité použitie `if` podmienok.
- **5** prác malo chybný alebo žiadny shebang.
- **2** študenti dostali doporučenie na použitie konštrukcie `.join()` pri spájaní stringov nachádzajúcich sa v liste.
- **2** práce obsahovali syntaktickú chybu.
- **2** študenti dostali doporučenie na použitie `if item in my_list:`, ak chceli porovnávať rovnakú premennú oproti väčšiemu počtu hodnôt.
- **1** práca dostala doporučenie na iterovanie cez štruktúru pomocou `for...in...`

Pri nasadení systému v praxi sa vyskytovali aj chyby. Chyby v testoch, na ktoré som bol upozornený študentami, boli opravené. Nanešťastie sa ale prejavila chyba pri kontrole plagiátov. Pri zvýšenom počte prác sa zvyšoval aj počet kontrol plagiátov a tento zvýšený počet vytvoril záťaž na systém, ktorá sa prejavila spomalením vyhodnotenia prác. Z toho dôvodu musela byť kontrola plagiátov v reálnom čase stiahnutá.

Je avšak vytvorený samostatný modul v súbore *plagiat.py*, ktorý je schopný kontrolovať plagiáty medzi dvoma špecifikovanými priechkami, ktorých cesty je potrebné definovať.

7.7 7. projekt

Siedmy projekt vyžadoval vypracovanie funkcií s prokročilými štruktúrami.

- `@limit_calls(max_calls, error_message_tail)` - tento dekorátor obecných funkcií má za úlohu zastaviť vykonávanie programu, ak sa dekorovaná funkcia zavolala viackrát, ako udáva parameter `max_calls`. Pri zastavení je vypísaná chybová správa definovaná podľa `error_message_tail`, ktorá môže mať pri vytváraní dekorátora aj defaultnú hodnotu.

Pri tomto dekorátore bola testovaná predovšetkým funkcionálnosť na základe počtu volaní, správnych chybových správ a použitie defaultných hodnôt aj pri samotnej definícii dekorátora. Tradične boli testované aj nesprávne vstupy, ktoré sa na základe zadania mohli vyskytnúť.

- `ordered_merge(*items, selector)` - generátorová funkcia, ktorá na základe hodnôt v parametri `selector` vyberá z iterovateľných objektov `*items` prvok na základe jeho indexu. Parametrov `*items` môže byť ľubovoľné množstvo.

Keďže táto funkcia mohla mať ľubovoľný počet parametrov, bolo veľa priestoru na chyby. Preto bolo veľa testov založených práve na chybových vstupoch. Taktiež sa mohlo stať, že pri zadaných hodnotách index vystúpil mimo poľa a na tento fakt by mal byť študent pripravený.

- `Log('my_log.txt')` - logovacia trieda, ktorá musí definovať metódu `logging(string)`, ktorá zapisuje `string` do triedou definovaného súboru. Bez ohľadu na počet volaní metódy `logging(string)` muselo byť v súbore napísané vždy na začiatku `Begin` a na konci `End`.

Keďže sa jednalo o prácu so súborom, bolo potreba vytvárať súbor s jedinečným menom pre každého študenta. Ako názov som použil login študenta. Tým pádom sa vylúčilo akékoľvek prepísanie logovacieho súboru niekým iným. V súbore *checks_proj07.py* bol predovšetkým kontrolovaný obsah logovacieho súboru. Trieda mala fungovať nasledovne:

```
with Log('mylog.txt') as logfile:
    logfile.logging('Test1')
    logfile.logging('Test2')
```

V rámci 7. projektu bolo otestovaných **43** študentských prác. Tento projekt bol taktiež testovaný v praxi. Plagiáty v tomto projekte systém nenašiel.

- **37** študentom systém doporučil použiť variantu `with open()` pri otváraní súboru namiesto `open()` no v rámci použitia v tomto projekte je vyžadovaná jednoduchšia verzia.
- **25** prác obsahovalo chybné alebo žiadne docstringy.
- **20** študentov nevyužilo *arbitrary argument* pri argumente `selector`, ktorého použitie bolo vhodné vo funkcii `ordered_merge()`. Mnohí namiesto toho použili `**kwargs`.
- **9** prác neobsahovalo jednu alebo viac zo zadaných konštrukcií. Dekorátor `limit_calls()`, generátorová funkcia `ordered_merge()` alebo triedu `Log` s metódou `logging()`.
- **7** študentov dostalo doporučenie na využívanie plného potenciálu `try-except` bloku. Mnohí ho využívali len ako pokus či daná konštrukcia prejde.
- **4** študenti dostali doporučenie použiť `return` na návrat operácii, ako len na návrat premennej.
- **4** práce nemali správne definované parametre dekorátora `limit_calls()`.
- **2** študenti používali porovnanie premenných ku *boolean* hodnotám. Bolo im doporučené porovnávať tieto premenná priamo: `if variable:` alebo `if not variable:`.
- **1** práca mala chybný alebo žiadny shebang.

Kapitola 8

Závěr

V rámci tejto bakalárskej práce som najprv preštudoval chyby, ktorých sa začínajúci programátori dopúšťajú, ako aj dôvody automatizácie výuky v programovaní. Veľkú časť štúdia zabrali aj správne praktiky používané v jazyku Python. Následne som preštudoval moderné trendy tejto automatizácie v praxi.

Na základe týchto zistení som najprv navrhol a následne implementoval systém pre automatické vyhodnocovanie študentských projektov. Tento systém je zameraný na navrhovanie zlepšení pre použitie idiomatických konštrukcií jazyka Python. Obsahuje taktiež kontrolu funkcionality projektov odovzdávaných do predmetu *ISJ* v akademickom roku 2017/2018. Na týchto projektoch bol aj systém testovaný. Celkovo bolo otestovaných 1063 študentských projektov a poskytnutých bolo 1517 doporučení na zlepšenie.

Výsledný systém hodnotím pozitívne. Je pripravený na použitie v praxi a ukázalo sa, že je schopný poskytnúť dostatočné množstvo doporučení pre študentov a je už len na nich, či sa z nich poučia. Začiatočným programátorom by určite pomohol vo výuke. Ku systému mám však aj dosť pripomienok. Nie všetko sa podarilo podľa plánu. Neskoré nasadenie do praxe malo za následok vytvorenie nedostatku času na úpravu systému. Na základe nasadenia som zistil, že modul na odhalenie plagiátov nezvládne veľké množstvo porovnaní projektov a tým pádom spomalil celý systém. To vytvorilo prekážku pri testovaní projektov v čase pred odovzdávaním. Taktiež mali niektoré testy na vyhľadanie idiémov nekonzistentné výsledky, ktoré sa do určitej miery podarilo opraviť. Systém teda hodnotím kladne ale určite by bolo čo zlepšovať.

Z dôvodu krátkeho času aktivity systému som sa rozhodol nezískavať spätnú väzbu od študentov. Kvôli spomínaným problémom a malej testovacej vzorke by výsledky boli zväčša negatívne.

Aplikáciu by bolo možné rozšíriť rôznymi spôsobmi. Jedným z nich by bola konzistentná kontrola plagiátov. Taktiež by bolo pre používateľa atraktívne, keby malo webové rozhranie lepši dizajn. Týmto aspektami som sa však pri vývoji zapodieval iba okrajovo. Ďalším zo spôsobov na zlepšenie je možnosť pridávať nové testy do systému v takom štádiu, v akom je. To znamená pridávať testy idiémov, ako aj testy projektov. So zvýšeným počtom testov by bol systém výkonnejší na použitie v rámci predmetu *ISJ*. Posledným, a pre mňa najzaujímavejším zo spôsobov je rozšíriť systém čisto na výuku pythonovských idiémov. Jednoduchým odstránením validácie loginu s pomenovaním príkladného súboru je možné systém použiť na vyhodnotenie akéhokoľvek skriptu v jazyku Python. Tým sa otvára brána použitia aj mimo predmet.

Literatúra

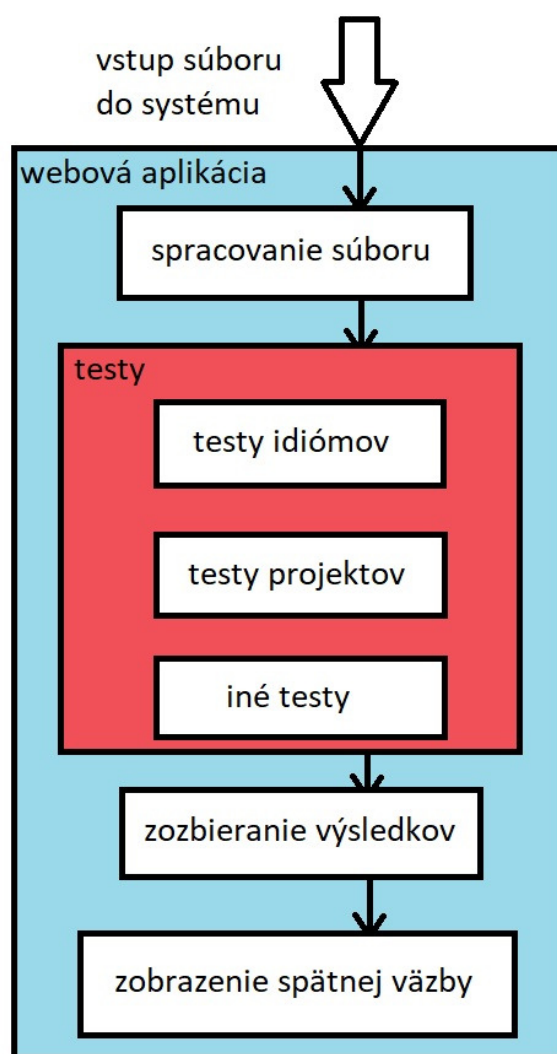
- [1] Altadmri, A.; Brown, N. C. C.: 37 Million Compilations: Investigating Novice Programming Mistakes in Large-Scale Student Data. [Online; navštíveno 11.01.2018]. URL <http://twistedsquare.com/37Million.pdf>
- [2] Butz, C. J.; Hua, S.; Maguire, R. B.: *A Web-Based Intelligent Tutoring System for Computer Programming*. IEEE, 2004, ISBN 0-7695-2100-2. URL <https://ieeexplore.ieee.org/abstract/document/1410798/>
- [3] Downey, A. B.: *Think Python: How to Think Like a Computer Scientist*. O'Reilly Media, Inc, 2015, ISBN 9781491939413. URL <https://books.google.cz/books?id=mZwbCwAAQBAJ&dq>
- [4] Drake, F. L.: Python 3.6.5 documentation. 2018, [Online; navštíveno 26.04.2018]. URL <https://docs.python.org/3/library/exceptions.html>
- [5] Fangohr, H.; O'Brien, N.: Teaching Python programming with automatic assessment and feedback provision. [Online; navštíveno 10.01.2018]. URL <https://arxiv.org/pdf/1509.03556.pdf>
- [6] Goodger, D.; van Rossum, G.: PEP 257 – Docstring Conventions. 2001, [Online; navštíveno 12.01.2018]. URL <https://www.python.org/dev/peps/pep-0257/>
- [7] Kernighan, B. W.; Pike, R.: *The Practice of Programming*. Addison-Wesley Professional, 1999, ISBN 9780133133417. URL https://books.google.cz/books?id=_KbfCQAAQBAJ&dq
- [8] Kluyver, T.: Green Tree Snakes - the missing Python AST docs. 2012, [Online; navštíveno 29.04.2018]. URL <https://greentreesnakes.readthedocs.io/en/latest/>
- [9] Knupp, J.: *Writing Idiomatic Python*. 2017. URL <https://jeffknupp.com/writing-idiomatic-python-ebook/>
- [10] Pea, R. D.: Language-Independent Conceptual 'Bugs' in Novice Programming. [Online; navštíveno 23.04.2018]. URL https://web.stanford.edu/~roypea/RoyPDF%20folder/A28_Pea_86.pdf
- [11] Ronacher, A.: Jinja2 Documentation. 2008, [Online; navštíveno 2.05.2018]. URL <http://mitsuhiko.pocoo.org/jinja2docs/Jinja2.pdf>

- [12] van Rossum, G.; Warsaw, B.; Conghlan, N.: PEP 8 – Style Guide for Python Code. 2001, [Online; navštíveno 12.01.2018].
URL <https://www.python.org/dev/peps/pep-0008/>
- [13] Saikkonen, R.; Malmi, L.; Korhonen, A.: *Fully Automatic Assessment of Programming Exercises*. ACM New York, 2001, ISBN 1-58113-330-8.
URL <https://dl.acm.org/citation.cfm?doid=377435.377666>
- [14] Shipman, J. W.: Python XML processing with lxml. 2013, [Online; navštíveno 1.05.2018].
URL <http://infohost.nmt.edu/tcc/help/pubs/pylxml/pylxml.pdf>
- [15] Singh, R.; Gulwani, S.; Solar-Lezama, A.: Automated Feedback Generation for Introductory Programming Assignments. [Online; navštíveno 6.05.2018].
URL <http://people.csail.mit.edu/rishabh/papers/autograderPLDI13.pdf>
- [16] Widera, M.: Why Testing Matters in Functional Programming. [Online; navštíveno 4.05.2018].
URL <http://www.cs.nott.ac.uk/~psznhn/TFP2006/Papers/05-Widera-WhyTestingMattersInFP.pdf>

Prílohy

Príloha A

Návrh



Obr. A.1: Schéma zjednodušeného pracovného toku systému.

Príloha B

Grafické užívateľské rozhranie




Obr. B.1: Domovská stránka webového rozhrania.



Obr. B.2: Stránka vyhodnotenia chýb.



Obr. B.3: Detailný popis doporučení PEP 8.



Testovať ďalší súbor

Naspäť

Praktiky kódu

```

1 #!/usr/bin/env python3
2
3 import re
4
5 def first_nonrepeating(string):
6     """
7     Funkcia vráti prvý neopakujúci sa znak.
8
9     :param string: vstupný reťazec
10    :type string: string
11
12    :returns: znak alebo nič
13    :rtype: char/None
14    """
15    i = 0
16    for char in string:
17        if char == ' ' or char == '\t':
18            continue
19        znak = char
20        for char in string:
21            if znak == char:
22                i = i + 1
23        if i == 1:
24            return znak

```

Line	Popis chyby
0,	Funkcia by vám pri neočakávanom vstupe nemala padať: <code>first_nonrepeating(65)</code> .
24,	Skúste využívať <code>return</code> na vyhodacovanie výrazov alebo funkcií, nie <code>len</code> na vracanie hodnôt von z funkcie. Porozmyšľajte či to nie je váš prípad.
94,	Ak je to vhodné, tak sa snažte využívať plný potenciál návratovej funkcie <code>raise</code> v <code>try-except</code> bloku, ktorá uchováva užitočné informácie o vyskytnutej chybe. Viac napríklad na: https://docs.python.org/3.6/tutorial/errors.html

Obr. B.4: Detailný popis návrhu praktík na zlepšenie.